F A C H H O C H S C H U L E W E D E L

B A C H E L O R A R B E I T

in der Fachrichtung

Wirtschaftsinformatik

iDocIt!:
Werkzeug zur einheitlichen Dokumentation für verschiedene Schnittstellenspezifikationen

Eingereicht von:

Dirk Meier-Eickhoff
Schulstraße 4, 21649 Regesbostel, Tel. (04165) 222 796

Erarbeitet im:	7. Semester
Abgegeben am:	01.03.2011
Referent (FH Wedel):	Prof. Dr. Sebastian Iwanowski Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel. (04103) 80 48-63
Referent (Praxis):	Jan Christian Krause AKRA GmbH Domstraße 17 20095 Hamburg Tel. (040) 309 535-30

Vorwort

Ich möchte mich ganz herzlich bei Herrn Prof. Dr. Sebastian Iwanowski und bei Herrn Jan Christian Krause für die sehr gute Betreuung dieser Arbeit bedanken. Die kreativen Diskussionen mit Herrn Krause haben mir in sehr vielen Fällen weitergeholfen.

Ganz besonderer Dank gilt meiner Familie und speziell meiner Frau, die während meines Studiums unablässig hinter mir standen. Ohne ihre Unterstützung wäre mein Studium nicht möglich gewesen.

Dirk Meier-Eickhoff, im März 2011

Inhaltsverzeichnis

Tabellenverzeichnis.....5

Abbildungsverzeichnis.....7

1 Einführung.....9
1.1 Motivation.....9
1.2 Ziele und Gang dieser Untersuchung.....11

2 Dokumentation von Operationen mit thematischen Rastern.....12

3 Anforderungsanalyse.....21
3.1 Allgemeine Schnittstellendokumentation.....22
3.2 Format.....23
3.3 Dokumentationswerkzeug.....25
3.4 Vorgaben der AKRA GmbH.....28

4 Existierende Dokumentationssysteme.....29
4.1 Kategorien von Dokumentationssystemen.....29
4.2 Unterstützung der Dokumentation mit thematischen Rollen.....31
4.3 Fazit.....36

5 Systementwurf.....37
5.1 Grammatiken.....38
5.1.1 Grundlagen formaler Grammatiken.....38
5.1.2 Generische Grammatik für Schnittstellen.....40
5.2 Modularisierung und Erweiterbarkeit.....48
5.2.1 Eclipse Plugins.....49
5.2.2 iDocIt!.....50
5.3 Parser für Schnittstellen.....56
5.4 User Interface (UI).....62
5.5 Kommunikation der Plattform-Services.....66

6 Implementierung.....70
6.1 Plattform (Core).....70
6.1.1 Abstrakte Schnittstellenstruktur.....70
6.1.2 Parser Extension Point.....75
6.2 Parser für WSDL.....76
6.2.1 Existierende XML / WSDL Parser.....77
6.2.2 Struktur der Dokumentationen in WSDL.....78
6.2.3 Implementierungsdetails des WSDL-Parsers.....80
6.3 Parser für Java.....87
6.3.1 Existierende Java Parser.....88
6.3.2 Struktur der Dokumentation in Javadoc.....89
6.3.3 Implementierungsdetails des Java-Parsers.....91
6.4 User Interface (UI).....97

7 Systembewertung.....103
7.1 Fallbeispiel.....103
7.1.1 Dokumentation mit iDocIt!.....104

7.1.2 Dokumentation mit Eclipse und Javadoc....109
7.2 Realisierung der Anforderungen....110
7.3 Probleme des Prototypen....112
7.4 Fazit....113

8 Abschlussbetrachtungen....116
8.1 Zusammenfassung und Ausblick....116
8.2 Mögliche Weiterentwicklungen....118

A Auswahl existierender Dokumentationssysteme....120

B WSDL 1.1 Spezifikation....126

C Java Interface Deklaration....128

D Benutzerhandbuch....131
D.1 Systemvoraussetzungen....131
D.2 Installation....132
D.3 Bedienungsanleitung....132
D.3.1 Konfiguration von iDocIt!....132
D.3.2 Erstellen von Dokumentationen....135

E Abkürzungsverzeichnis....139

Literaturverzeichnis....141

Eidesstattliche Erklärung....145

Tabellenverzeichnis

Tabelle 2.1: Auszug aus der Schnittstellenbeschreibung der Operation GetFeedback aus der eBay Trading API.....14
Tabelle 2.2: Beispiel eines thematischen Rasters für das Verb to get im Kontext der Operation GetFeedback aus der eBay Trading API.....15
Tabelle 2.3: Auszug aus der Schnittstellenbeschreibung der Operation GetFeedback aus der eBay Trading API mit beispielhafter Zuordnung von thematischen Rollen.....17
Tabelle 3.1: Auswahl an möglichen Adressaten.....23
Tabelle 3.2: Enumerationswerte von DetailLevelCodeType aus der eBay Trading API 26
Tabelle 3.3: Auszug aus der Schnittstellenbeschreibung der Operation GetFeedback aus der eBay Trading API.....26
Tabelle 4.1: Unterstützte Dokumentationstypisierungen von den untersuchten Kategorie 3 und 4 Dokumentationssystemen.....32
Tabelle 4.2: Fortsetzung: Unterstützte Dokumentationstypisierungen von den untersuchten Kategorie 3 und 4 Dokumentationssystemen.....33
Tabelle 4.3: Unterstützung nicht sichtbarer Elemente von den untersuchten Kategorie 3 und 4 Dokumentationssystemen.....34
Tabelle 4.4: Korrelation zwischen Erweiterbarkeit und Anzahl unterstützter Programmier- und Beschreibungssprachen von den untersuchten Dokumentationssystemen.....35
Tabelle 5.1: Beispiel einer Grammatik G.....39
Tabelle 5.2: Auszug der WSDL 1.1 Spezifikation für das <portType>-Element,.....41
Tabelle 5.3: Allgemeine Grammatik einer Schnittstellenstruktur in EBNF (vereinfachte Darstellung).....45
Tabelle 5.4: Inhalt der plugin.xml für die WSDL Parser-Erweiterung.....57
Tabelle 5.5: Pfad von der WSDL Message GetFeedbackRequest der Operation GetFeedback aus der eBay Trading API bis zum Attribut UserID vom Typ UserIDType.....60
Tabelle 6.1: Auszug der Klasse Parameters.....74
Tabelle 6.2: Auszug der Klasse SignatureElement.....74
Tabelle 6.3: Auszug der Klasse WSDLMessage.....75
Tabelle 6.4: <documentation>-Elementstruktur für Dokumentation von thematischen Rollen und verschiedenen Adressaten.....79
Tabelle 6.5: Auszug von WSDL Elementen mit den von WSDL4J unterstützten Attributen.....81
Tabelle 6.6: Auszug aus WSDL 1.1 Spezifikation,.....82
Tabelle 6.7: Zuordnung der iDocIt!-Dokumentationsblöcke in die WSDL der eBay Trading API.....86
Tabelle 6.8: Auszug der Java Sprachspezifikation für eine Interface Deklaration in EBNF.....92
Tabelle 6.9: Auszug der möglichen Modifier der Java Sprachspezifikation in EBNF 93
Tabelle 6.10: Beispiel einer Java-Klasse zur Erläuterung der JavaBeans Spezifikation 93
Tabelle 6.11: Auszug der Methode aus der Klasse DocumentItemListComposite zum Hinzufügen einer Documentation (DocumentItemComposite).....101

Tabelle 6.12: Auszug der Methode aus der Klasse DocumentItemListComposite zum Entfernen einer Documentation.....102
Tabelle 7.1: Definierte thematische Raster für das Fallbeispiel.....105
Tabelle 7.2: Javadoc-Raster für getFeedback().....110
Tabelle 7.3: Ausgefülltes Javadoc-Raster für getFeedback().....110
Tabelle A.1: Kategorien und allgemeine Informationen zu den untersuchten Dokumentationssystemen.....120
Tabelle A.2: Unterstützte Dokumentationstypisierungen von den untersuchten Dokumentationssystemen.....121
Tabelle A.3: Fortsetzung: Unterstützte Dokumentationstypisierungen von den untersuchten Dokumentationssystemen.....122
Tabelle A.4: Unterstützung nicht sichtbarer Elemente von den untersuchten Dokumentationssystemen.....123
Tabelle A.5: Unterstützte Programmier- und Beschreibungssprachen von den untersuchten Dokumentationssystemen.....124
Tabelle A.6: Links zu den untersuchten Dokumentationssystemen.....125
Tabelle B.1: WSDL 1.1 Spezifikation.....126
Tabelle B.2: Fortsetzung: WSDL 1.1 Spezifikation.....127
Tabelle C.1: Auszug der Java Sprachspezifikation (3. Edition) für eine Interface Deklaration in EBNF.....130

Abbildungsverzeichnis

Abbildung 2.1: Rauschen in der O-Doku am Beispiel des Eingabeparameters ItemID der Operation GetFeedback aus der eBay Trading API.....13
Abbildung 2.2: Beispielhafte Kürzung der Dokumentation für die Operation GetFeedback aus der eBay Trading API durch Zuordnung thematischer Rollen.....18
Abbildung 2.3: Rauschen in der O-Doku am Beispiel des Eingabeparameters FeedbackID der Operation GetFeedback aus der eBay Trading API 19
Abbildung 2.4: Beziehungen thematischer Rollen zu Verbklassen.....20
Abbildung 3.1: Aufbau eines Dokumentationsblocks.....24
Abbildung 4.1: Kategorien von Dokumentationssystemen.....29
Abbildung 5.1: Aufteilung in fixe und variable Komponenten von iDocIt! im Kontext von Eclipse.....37
Abbildung 5.2: Beispielhafter Aufbau eines <portType>-Elements der WSDL 1.1 Spezifikation.....42
Abbildung 5.3: Die Klasse Parameter mit den zusammengefassten Attributen.....43
Abbildung 5.4: Klassendiagramm der allgemeinen Grammatik einer Schnittstellenstruktur.....46
Abbildung 5.5: Auszug der abstrakten Klasse SignatureElement.....47
Abbildung 5.6: Komponentendiagramm von iDocIt!.....53
Abbildung 5.7: Einordnung der iDocIt! Komponenten in die Drei-Schichten-Architektur.....54
Abbildung 5.8: Klassendiagramm der Klasse ThematicGridService.....54
Abbildung 5.9: Klassendiagramm der Klasse PersistenceService.....54
Abbildung 5.10: Klassendiagramm des Interfaces Parser.....55
Abbildung 5.11: Komponentendiagramm der Parser Extension (siehe auch Abbildung 5.6).....56
Abbildung 5.12: Klassendiagramm des Interfaces Parser mit seinen Kindklassen....57
Abbildung 5.13: Beispielhafte Abbildung einer WSDL 1.1 Struktur in die allgemeine Schnittstellenstruktur.....58
Abbildung 5.14: Layoutentwurf des User Interfaces.....63
Abbildung 5.15: Beispiel für die Ermittlung von zugeordneten thematischen Rollen 65
Abbildung 5.16: Klassendiagramm der UI Komponenten.....65
Abbildung 5.17: Aktivitäten beim Start von Eclipse.....66
Abbildung 5.18: Ablauf der Methodenaufrufe, beim Öffnen einer Datei in iDocIt!..67
Abbildung 5.19: Aktivitätsdiagramm zum Funktionsablauf, wenn ein Benutzer ein Signaturelement im UI selektiert.....68
Abbildung 5.20: Ablauf der Methodenaufrufe, beim Speichern von Änderungen.....69
Abbildung 6.1: Beispiel der Wiederverwendung des Datentyps Customer.....71
Abbildung 6.2: Dokumentationsblock im iDocIt! UI.....79
Abbildung 6.3: Klassen der allgemeinen Schnittstellenstruktur für WSDL.....83
Abbildung 6.4: Anzeige der Struktur der Operation GetFeedback aus der eBay Trading API im iDocIt! UI.....85
Abbildung 6.5: Javadoc Beispiel in HTML mit iDocIt! Dokumentationsblöcken.....91
Abbildung 6.6: Klassen der allgemeinen Schnittstellenstruktur für Java.....94

Abbildung 6.7: UI Klassen mit Methodennamen (keine vollständige Methodensignatur).....98
Abbildung 6.8: Auszug der Klasse DocumentItemListCompositeSelection.....99
Abbildung 7.1: Auszug der Klasse GetFeedbackCall.....104
Abbildung 7.2: Screenshot der Klassenstruktur von GetFeedbackCall in iDocIt!…105
Abbildung 7.3: In iDocIt! dokumentierte Methode GetFeedback() der Klasse GetFeedbackCall.....107
Abbildung 7.4: HTML-Ansicht der iDocIt! Dokumentation für die Methode GetFeedback.....109
Abbildung 7.5: Realisierte Anforderungen im UI.....112
Abbildung D.1: iDocIt! Preference Page.....133
Abbildung D.2: iDocIt! Preference Page für Adressaten.....133
Abbildung D.3: iDocIt! Preference Page für thematische Raster.....134
Abbildung D.4: iDocIt! Preference Page für thematische Rollen.....135
Abbildung D.5: iDocIt! Editor mit nicht unterstützter Datei geöffnet.....136
Abbildung D.6: iDocIt! Editor mit der geöffneten Klasse GetFeedbackCall aus dem eBay SDK for Java.....137

1 Einführung

1.1 Motivation

Die Erstellung und Pflege von Software-Dokumentation ist ein wichtiger und in der Praxis meistens aufwendiger Prozess. Dies gilt insbesondere für die Dokumentation von Schnittstellen. Eine fehlende Dokumentation erschwert die Einarbeitung und Nutzung erheblich. Ohne Dokumentation bleibt einem Konsumenten einer Schnittstelle nur das Lesen des Quelltextes der Implementierung oder Ausprobieren verschiedener Eingaben, um deren Nutzungsweise zu erfahren. Erstere Möglichkeit entfällt häufig, weil der Quelltext der Implementierung nicht immer verfügbar ist, wie z. B. bei Webservices oder Schnittstellen proprietärer Systeme. Daher sollte die präzise, knappe, verständliche und vollständige Dokumentation der Schnittstelle und ihrer Operationen angestrebt werden.[1,2,3]

Die Dokumentation von Schnittstellen wird in Projekten meist stiefmütterlich behandelt. Oft sind sie unvollständig, teils auch schwer verständlich und nicht einheitlich gestaltet. In vielen Fällen liegt dies am konzeptionellen Aufwand, sowie fehlenden stilistischen oder inhaltlichen Vorgaben.

Um festzustellen ob eine Operationsdokumentation (O-Doku) vollständig ist, bekommt der Autor nur rudimentäre Unterstützung von Werkzeugen. In der Regel wird ein Raster der öffentlich sichtbaren Elemente der Operationssignatur vorgegeben, z. B. die formalen Parameter oder Rückgabewerte. Der äußerliche Stil wird zwar durch durchgängig genutzte Formate wie Javadoc vereinheitlicht, aber die inhaltliche Qualität der Dokumentation variiert je nach Autor.

1 Vgl. Clements, Paul et al.: Documenting Software Architectures, 2008, S. 228 - 239
2 Vgl. Schraitle, Thomas: Value of (Good) Documentation, 2009
3 Vgl. Reilly, Annette D. et al.: IEEE Std 1063-2001, 2001, S. 5ff

Die Kunden der AKRA GmbH verlangen zunehmend die qualitativ hochwertige Dokumentation der zu entwickelnden Software. Immer häufiger ist die Einhaltung von entsprechenden Standards (z. B. IEEE Std 1063[4] oder ISO/IEC 12207[5]) vertraglich festgeschrieben. Die AKRA GmbH gewährleistet dies aktuell hauptsächlich durch Schulungen der Mitarbeiter. Allerdings bleiben erfahrungsgemäß weiterhin qualitative Unterschiede der Dokumentation verschiedener Autoren bestehen.

Um die Erstellung qualitativ hochwertiger Dokumentation zu standardisieren und zu beschleunigen, unterstützt die AKRA GmbH das Promotionsvorhaben von Herrn Krause, dem Praxis-Referenten dieser Arbeit. Herr Krause entwickelt ein Konstruktionsverfahren für API-Dokumentation, welches auf Basis des Operationsbezeichners semantische Lücken in der Dokumentation identifizieren und bei deren Füllung assistieren soll.

Es fehlt derzeit noch ein implementiertes Werkzeug, welches die Mitarbeiter bei der Anwendung dieses Ansatzes unterstützt. Des weiteren plant AKRA sich durch die öffentlichkeitswirksame Präsentation dieses Werkzeugs in Form eines Open Source-Projektes als Innovator im Umfeld der API-Dokumentation zu positionieren. So will sich die AKRA GmbH einen Wettbewerbsvorteil gegenüber ihren Mitbewerbern im Projektgeschäft verschaffen.

4 Vgl. Reilly, Annette D. et al.: IEEE Std 1063-2001, 2001
5 Vgl. o.V.: ISO/IEC 12207:2008, 2008

1.2 Ziele und Gang dieser Untersuchung

Im Rahmen dieser Bachelorarbeit wird untersucht, wie ein Autor beim Verfassen von O-Doku technisch unterstützt werden kann. Grundlage für diese Untersuchung ist der Ansatz von Krause[6,7], Operationen mit Hilfe von thematischen Rastern zu dokumentieren. Zunächst wird der Ansatz von Krause beschrieben und gezeigt wie damit Operationen dokumentiert werden können. Daraus werden Anforderungen an Dokumentationswerkzeuge und -formate abgeleitet und existierende Werkzeuge und Formate auf diese hin evaluiert.

Da kein Werkzeug alle Anforderungen erfüllt, wird ein neues prototypisches Werkzeug mit dem Namen iDocIt! entworfen und implementiert.

Anschließend werden Potential und Lücken des neu entwickelten Werkzeugs anhand eines Fallbeispiels gezeigt. Diese Arbeit schließt mit einer Zusammenfassung und einem Ausblick, sowie einigen möglichen Weiterentwicklungen, die u. a. die tägliche Arbeit mit iDocIt! erleichtern sollen.

6 Vgl. Krause, Jan Christian: Thematic Grids, 2010
7 Vgl. Krause, Jan Christian: Stille und Rauschen, 2011

2 Dokumentation von Operationen mit thematischen Rastern

In der Praxis werden in der Regel die öffentlich sichtbaren Elemente einer Operationssignatur (z. B. Eingaben, Ausgaben und Ausnahmen bzw. Fehler) dokumentiert. Eine O-Doku gilt als vollständig, wenn jedes Element einmal dokumentiert ist. Wichtige Bestandteile des Schnittstellenvertrags bleiben so undokumentiert.[8] Zum Beispiel werden Seiteneffekte und verwendete Ressourcen, die nicht als Parameter deklariert sind, nicht berücksichtigt. Der Autor muss selbst die Operation nach Seiteneffekten und implizit verwendeten Ressourcen analysieren, um sie dann zu dokumentieren. Dieses birgt die Gefahr, dass etwas übersehen wird und damit undokumentiert bleibt. Diese Problematik lässt sich durch die Metaphern Stille und Rauschen charakterisieren[9,10]:

- **Stille:** Relevanter Aspekt der Operation, welcher in der zugehörigen Dokumentation nicht erwähnt wird
- **Rauschen:** Redundante bzw. unnötig ausführliche Dokumentation zu einem Aspekt der Operation

Wo Stille in der O-Doku auftreten kann, wurde gerade gezeigt. Wie Rauschen auftreten kann, wird an einem realen Beispiel aus der öffentlichen eBay Trading API [11] verdeutlicht.

In der eBay Trading API wird die Bewertung z. B. eines Benutzers, eines Artikels oder einer Transaktion Feedback genannt. Über die Operation `GetFeedback` kann die jeweilige Bewertung abgerufen werden. Diese Operation wollen wir im Folgenden näher betrachten.

8 Vgl. Züllighoven, Heinz: Object-Oriented Construction Handbook, 2005, S. 38ff
9 Vgl. Meyer, Bertrand: Formalism in Specifications, 1985, S. 6 - 25
10 Vgl. Krause, Jan Christian: Stille und Rauschen, 2011
11 Vgl. o.V.: eBay Trading API, o.J.

In Tabelle 2.1 ist die Dokumentation zur Operation `GetFeedback` auszugsweise abgebildet. Für die vier Eingabeparameter ist auch jeweils der erste Satz der Dokumentation aufgeführt, um zu zeigen, dass die Bezeichner noch einmal in Prosa wiedergegeben wurden, was eine Art von Rauschen ist. In Abbildung 2.1 wird dieses am Beispiel des Parameters `ItemID` gezeigt.

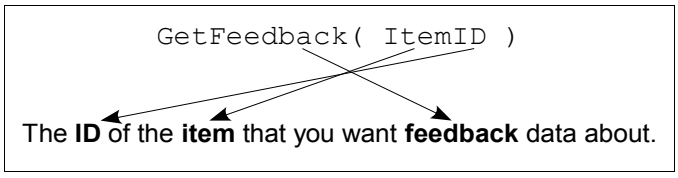


Abbildung 2.1: Rauschen in der O-Doku am Beispiel des Eingabeparameters `ItemID` der Operation `GetFeedback` aus der eBay Trading API

Die in Abbildung 2.1 gezeigte Beschreibung enthält keine neuen Informationen, denn der Bezeichner selbst `ItemID` sagt, dass es die ID eines Artikels ist und der Operationsbezeichner `GetFeedback` besagt, dass die Bewertung abgerufen werden soll. Weiterhin impliziert die ID im Bezeichner eine eindeutige Identifikation eines Benutzers. Die Beschreibung ist also unnötig. Ein Kommentar soll über diese Worte hinausgehen und dem Leser Informationen offenbaren, die nicht sofort erkennbar sind. Es soll keine unnötigen Informationen enthalten.[12,13]

Um das Problem der Stille und des Rauschens anzugehen, wird ein neuer Dokumentationsansatz zur Erstellung von O-Doku verwendet. Dabei handelt es sich um eine verbbasierte Analyse von Operationsbezeichnern mit Ableitung von thematischen Rastern, die aus der Linguistik bekannt sind.[14]

12 Vgl. o.V: Javadoc, o.J., „A Style Guide“
13 Vgl. Martin, Robert C.: Clean Code, 2009, S. 61 - 71
14 Vgl. Krause, Jan Christian: Thematic Grids, 2010

Ein Satz wird in der natürlichen Sprache hauptsächlich durch das verwendete Verb beschrieben. Akteure oder Attribute einer Handlung sind dabei syntaktisch und semantisch abhängig von dem Verb. Die Akteure und Attribute werden auch Argumente des Verbs genannt, die obligatorisch oder optional sein können.[15,16]

Schnittstellenelement		**Dokumentation**
Operation:	`GetFeedback`	Use this call to retrieve the accumulated feedback left for a specified user, [...]
Eingabeparameter:		
`string`	`FeedbackID`	An ID that uniquely identifies a feedback record to be retrieved. [...]
`ItemIDType (string)`	`ItemID`	The ID of the item that you want feedback data about. [...]
`string`	`TransactionID`	Transaction ID whose feedback record you want to inspect. [...]
`UserIDType (string)`	`UserID`	Specifies the user whose feedback data is to be returned. […]
Ausgabe:		
`int`	`FeedbackScore`	Indicates the total feedback score for the user. [...]

Tabelle 2.1: Auszug aus der Schnittstellenbeschreibung der Operation `GetFeedback` *aus der eBay Trading API*[17]

In den Programmiersprachen hat sich wegen der starken Aussage des Verbs etabliert, dass ein Operationsbezeichner mit einem Verb beginnt.[18,19] Die Operation in Tabelle 2.1 ist ein Beispiel dafür. Die aus der Konvention resultierenden Bezeichner haben den Charakter eines Imperativsatzes (Satz in Befehlsform). Unter Einbeziehung des gegebenen Eingabeparameter `UserID` zum Beispiel:

get the *feedback* by *UserID*.

15 Vgl. Gruber, J. S.: Lexical Relations, 1965
16 Vgl. Fillmore, C. J.: Case for Case, 1968, S. 1 - 90
17 Vgl. o.V.: eBay Trading API, o.J., „GetFeedback“
18 Vgl. Martin, Robert C.: Clean Code, 2009, „Verbs and Keywords“, S. 43
19 Vgl. Krause, Jan Christian: Thematic Grids, 2010

Der Formalismus, um die Argumente eines Verbs zu bestimmen, heißt thematisches Raster (Thematic Grid). Die benötigten Argumente heißen thematische Rollen (Thematic Role)[20,21]. Ein vereinfachtes thematisches Raster für das Verb *get* im Kontext der Operation `GetFeedback` ist in Tabelle 2.2 abgebildet.

Thematisches Raster für:	to get
AGENT [obligatorisch]:	Ausführende Instanz
ACTION [obligatorisch]:	Eine bestimmte Information abrufen
OBJECT [obligatorisch]:	Die gesuchte Information
CRITERION oder PRIMARY KEY [optional]:	(Eindeutiges) Kriterium, welches angibt, ob das aktuell geprüfte OBJECT das gesuchte ist
SOURCE [optional]:	Quelle bzw. Speicher, aus dem die durchsuchten OBJECTs stammen
FORMULA [optional]:	Formel zur Berechnung der Information

Tabelle 2.2: Beispiel eines thematischen Rasters für das Verb to get im Kontext der Operation `GetFeedback` aus der eBay Trading API

Aus dem thematischen Raster kann man ablesen, dass *get* mehrere Argumente benötigt. Zum einen die Aktion (ACTION) selbst, bei der ein OBJECT abgerufen werden soll. Zudem ein Vergleichskriterium (CRITERION) bzw. ein eindeutiger Schlüssel (PRIMARY KEY), das zum Finden des richtigen OBJECTs verwendet werden soll. Außerdem werden eine Quelle (SOURCE) von wo die OBJECTs stammen und ggf., wenn das OBJECT berechnet wird, eine Formel (FORMULA) benötigt. Die Rolle AGENT ist nur implizit gegeben, die z. B. der Benutzer, das aufrufende Programm oder in der objektorientierten Welt das zugehörige Objekt einnehmen kann. Die Rollen AGENT, ACTION und OBJECT sind in diesem Beispiel obligatorisch und die Rollen CRITERION, SOURCE und FORMULA optional.

20 Vgl. Gruber, J. S.: Lexical Relations, 1965
21 Vgl. Fillmore, C. J.: Case for Case, 1968, S. 1 - 90

Wenn man die thematischen Rollen den Elementen der Operationssignatur zuordnet, bemerkt man, dass die Rollen

- SOURCE und
- FORMULA

keinem Signaturelement zugeordnet werden können (siehe Tabelle 2.3). Man hat dadurch eine implizit verwendete Ressource und eine wichtig zu dokumentierende Eigenschaft der Operation erkannt und das ohne die Implementierungsdetails zu kennen. Hierbei handelt es sich um nicht sichtbare Elemente, die aber zum Vertrag der Schnittstelle bzw. Operation gehören. Für diese Elemente herrscht häufig Stille in der O-Doku, weil diese von Dokumentationswerkzeugen (z. B. Javadoc) nicht im Raster zur Dokumentation zur Verfügung gestellt werden. Wenn die SOURCE als Parameter in die Operation hineingehen würde, wäre es ein in der Signatur explizit angegebenes Element und würde im Javadoc-Raster mit aufgeführt werden. Aber da Datenbankverbindungen u. ä. in der Regel im Hintergrund verwaltet werden, ist diese Vorgehensweise in der Praxis eher selten anzutreffen. Die SOURCE sollte aber dokumentiert werden, weil das Ergebnis der Operation maßgeblich von der implizit verwendeten Ressource abhängt. Beispielsweise bietet eBay Entwicklern neben dem Produktivsystem auch eine simulierte Sandbox-Umgebung[22] zum Testen ihrer entwickelten Programme an. Weiterhin sollte auch die Berechnungsformel (FORMULA) für die Bewertungskennzahl in der O-Doku enthalten sein. Sie beschreibt das Ergebnis der Operation und soll auch als Spezifizierung des Soll-Verhaltens dienen (siehe Kapitel 1.1). In der Dokumentation der eBay Trading API für `GetFeedback` ist sie aber nicht enthalten, sondern lediglich in der Benutzerhilfe von eBay[23].

22 Vgl. o.V.: eBay Developer's Sandbox, o.J.
23 Vgl. o.V.: eBay help: How Feedback works, o.J., „How are Feedback scores calculated?“

Schnittstellenelement		**Thematische Rolle**
Schnittstelle:	`eBayAPIInterface`	AGENT
Operation:	`GetFeedback`	ACTION
	?	SOURCE
	?	FORMULA
Eingabeparameter:		
`string`	`FeedbackID`	PRIMARY KEY
`ItemIDType (string)`	`ItemID`	PRIMARY KEY
`string`	`TransactionID`	PRIMARY KEY
`UserIDType (string)`	`UserID`	PRIMARY KEY
Ausgabe:		
`int`	`FeedbackScore`	OBJECT

Tabelle 2.3: Auszug aus der Schnittstellenbeschreibung der Operation `GetFeedback` aus der eBay Trading API[24] mit beispielhafter Zuordnung von thematischen Rollen

Durch die Zuordnung der thematischen Rollen können also nicht sichtbare Elemente aufgezeigt werden und so potentielle Stille in der O-Doku vermieden werden. Darüber hinaus können lange Beschreibungen durch die Zuordnung von Rollen eingespart werden. Die in Tabelle 2.3 beschriebenen Eingabeparameter sind jeweils eigenständige Primärschlüssel für verschiedene Arten von gesuchten Objekten. Durch die Zuordnung der Rolle PRIMARY KEY, dem Parameter- und dem Operationsbezeichner ist der Eingabeparameter aussagekräftig beschrieben. Der jeweilige erste Satz der Dokumentation ist somit redundant geworden und kann gestrichen werden.

24 Vgl. o.V.: eBay Trading API, o.J., „GetFeedback“

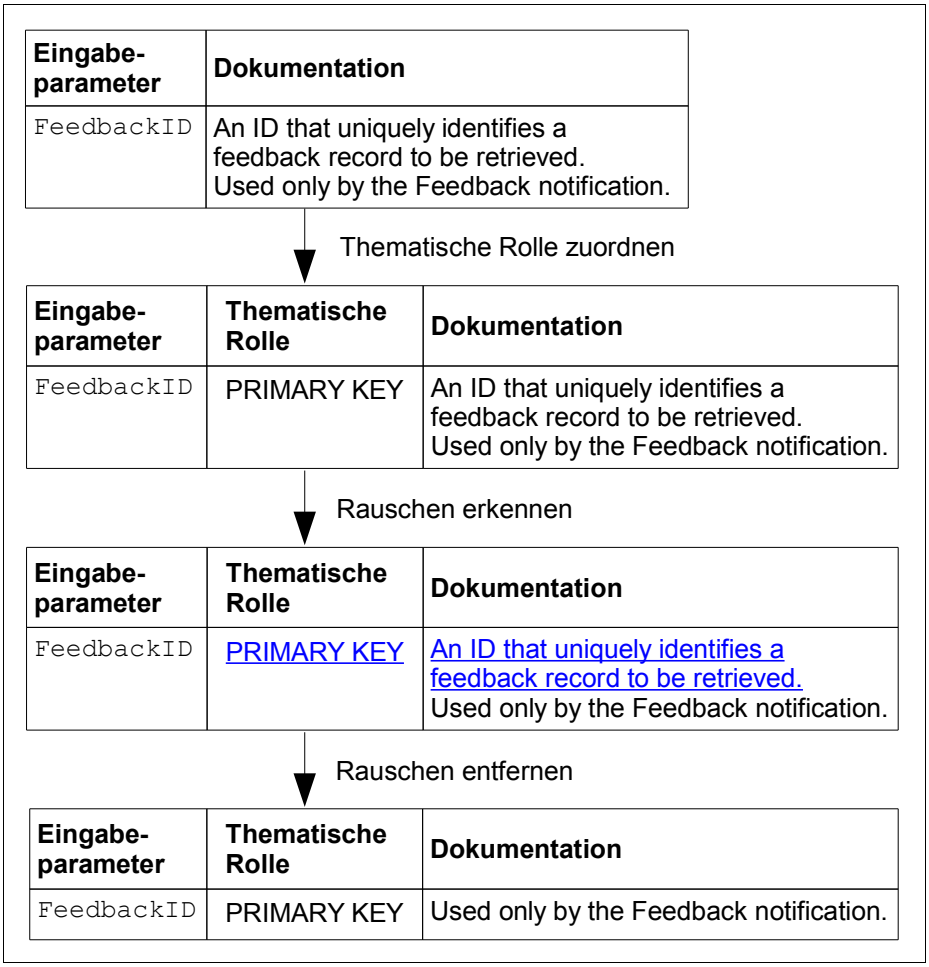


Abbildung 2.2: Beispielhafte Kürzung der Dokumentation für die Operation `GetFeedback` *aus der eBay Trading API durch Zuordnung thematischer Rollen*

Abbildung 2.2 zeigt diesen Vorgang beispielhaft. Durch die Zuordnung der thematischen Rolle ist eine Redundanz in der O-Doku hergestellt worden. Der erste Satz hat die gleiche Bedeutung wie die Rolle (siehe Markierung in Abbildung 2.2). Da einzelne Schlagwörter für einen Leser besser zu erfassen sind, als ein oder mehrere beschreibende Sätze, wird der redundante Satz gestrichen und die thematische Rolle bleibt. Die Aussage der O-Doku hat sich nicht verändert, ist aber kürzer und besser erfassbar geworden. In diesem speziellen Beispiel ist der erste Satz von sich aus schon als Rauschen einzustufen, da er den Bezeichner noch einmal ausformuliert wiedergibt (siehe Abbildung 2.3). Da er keine neuen Informationen

enthält, kann er einfach gestrichen werden. Eine Dokumentation soll nicht den Quelltext wiedergeben, sondern nicht offensichtliche Informationen beschreiben.

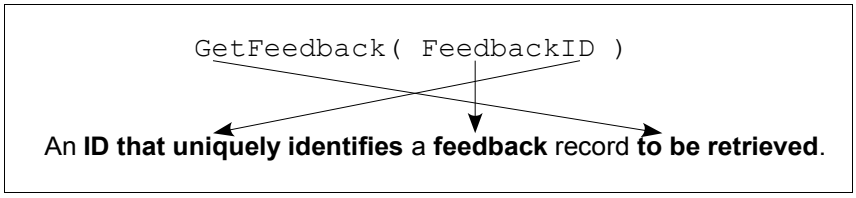


Abbildung 2.3: Rauschen in der O-Doku am Beispiel des Eingabeparameters `FeedbackID` *der Operation* `GetFeedback` *aus der eBay Trading API*

Wie weiter oben schon angedeutet, zeigt Tabelle 2.2 ein thematisches Raster für das Verb *get* in einer bestimmten Anwendungssituation (Kontext). In einem anderen Kontext wären vielleicht andere thematische Rollen passender, wobei einige Rollen (AGENT, ACTION und OBJECT) immer bestehen bleiben würden, da sie die obligatorischen Argumente des Verbs sind. Eigentlich müsste für jedes Verb in jedem möglichen Kontext ein thematisches Raster definiert werden, um alle möglichen Fälle abbilden zu können. Praktisch ist dieses aber aber nur mit großem Aufwand realisierbar. Daher sollten Verben, die häufig in bestimmten Kontexten verwendet werden, in Klassen aufgeteilt werden. Das hat den Vorteil, dass nicht für viele einzelne Verben ein thematisches Raster definiert werden muss, sondern für wenige Klassen von Verben. Eine Klasse enthält z. B. Verben die im Kontext einer Suche verwendet werden (siehe Abbildung 2.4):

Searching verbs: find, search, seek, get.

Die thematischen Rollen, die man dieser Klasse zuordnen kann, wären beispielsweise:

AGENT, ACTION, OBJECT, CRITERION, SOURCE.

Da viele Verben in unterschiedlichen Kontexten verwendet werden, ist es natürlich auch möglich, dass ein Verb gleichzeitig verschiedenen Klassen zugeordnet ist. Dann müsste man sich bei der Dokumentation abhängig von dem verwendeten Kontext für eines der definierten thematischen Raster zu dem Verb entscheiden. Das Verb *get* ist ein gutes Beispiel dafür, da es in fast allen Kontexten verwendet werden kann.

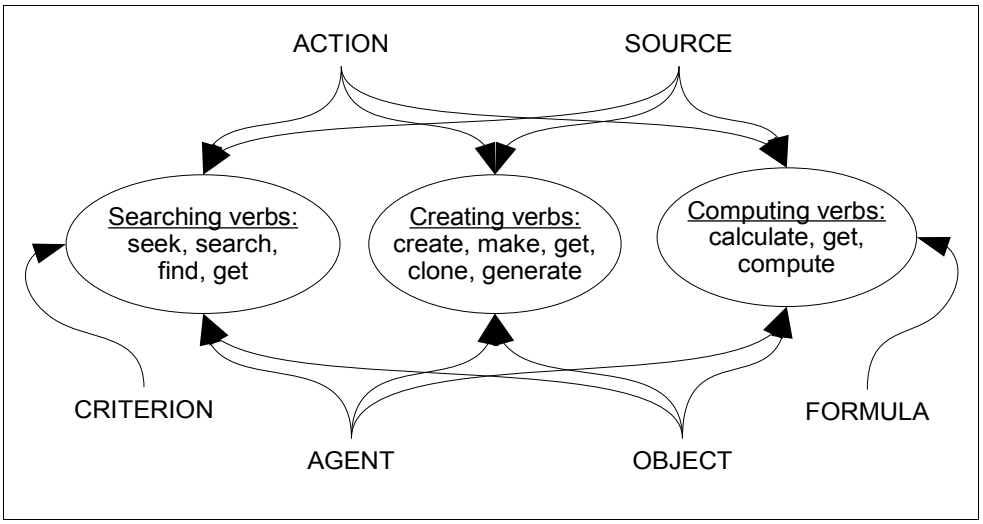


Abbildung 2.4: Beziehungen thematischer Rollen zu Verbklassen

Um für einen Benutzer den Aufwand für die Definition von thematischen Rastern noch weiter einzuschränken, kann er sich auf die Kontexte und Verben beschränken, die er in seinen Quelltexten verwendet. Dadurch entstehen domänen- bzw. unternehmensspezifische Raster.

3 Anforderungsanalyse

In dem vorherigen Kapitel haben wir gesehen, wie man eine Operation mit einem thematischen Raster dokumentiert. Dafür ergeben sich bestimmte Anforderungen, die ein Dokumentationswerkzeug und ein Format erfüllen müssen. In diesem Kapitel wird ermittelt, welche allgemeinen Basisanforderungen Schnittstellen-dokumentationen und welche softwaretechnischen Anforderungen ein Werkzeug erfüllen soll.

Die Formate von Dokumentationen verwenden zur Strukturierung Auszeichnungs-sprachen (Markup Language) bzw. Beschreibungssprachen (Descriptive Markup Language). Durch den strukturierten Aufbau können diese einfacher programmatisch ausgewertet werden. Eine Beschreibungssprache besitzt eine Syntax, durch die Informationen beschrieben werden können. Es existieren also Auszeichnungen (Markups), die einen logischen Zusammenhang zwischen einer Auszeichnung und einem Text herstellen.[25] In den Schnittstellendokumentationen werden z. B. Schlüsselwörter oder XML-Syntax als Auszeichnung und Strukturierung der Texte verwendet. Die Schlüsselwörter können verschiedene Ausprägungen haben. Bei Javadoc z. B. durch Wörter, die mit einem @-Zeichen beginnen (z. B. `@param`, um den logischen Zusammenhang zu einer Parameterbeschreibung herzustellen) und nach dem Schlüsselwort folgt der dazugehörige Text. In XML wird ein Text nicht nur durch ein Schlüsselwort eingeleitet, sondern durch einen Anfang und ein Ende umschlossen, z. B. `<Schlüsselwort>`Text`</Schlüsselwort>`.

In dem Kontext dieser Bachelorarbeit werden nur Dokumentationswerkzeuge behandelt, die bei der Erstellung und Bearbeitung von O-Doku bzw. Schnittstellen-dokumentationen unterstützen. Texteditoren, Editoren zur Erstellung von Modellen (z. B. UML) u. ä. werden hier nicht behandelt.

25 Vgl. Sebestyen, Thomas J.: XML, 2010, S. 26ff

3.1 Allgemeine Schnittstellendokumentation

Clements et al. und auch der IEEE Std 1063-2001 bieten einen umfassenden Katalog an Elementen einer Schnittstelle, die dokumentiert werden sollten.[26,27] Es wird auch beschrieben, welche Informationen in den Dokumentationen enthalten sein sollten. Ein Teil dieses Katalogs, der für den betrachteten Aspekt sinnvoll ist, soll als Basisanforderung an Dokumentationsformate und -werkzeuge dienen:

- Eine Schnittstelle soll mit Metadaten wie z. B. einer Versionsnummer und ggf. weiteren verwendeten Schnittstellen dokumentiert werden können.
- Von der Schnittstelle zur Verfügung gestellte Operationen sollen mit ihrer Syntax (z. B. Parameter) und Semantik beschrieben werden können. Es soll dazu auch dokumentiert werden können was passiert, wenn sie verwendet werden, mit allen Einschränkungen bei der Verwendung.
- Die Dokumentation verwendeter Datentypen soll referenziert werden können.
- Potentiell auftretende Fehler und Ausnahmen sollen dokumentiert werden können.
- Vorhandene Konfigurationsmöglichkeiten und ihre Auswirkungen sollen dokumentiert werden können.
- Von der Schnittstelle benötigte Ressourcen (z. B. Eingabeparameter, Dateien oder Datenbankverbindungen) sollen, wie die von ihr zur Verfügung gestellten Operationen, in ihrer Syntax, Semantik, Verwendung und Einschränkungen dokumentiert oder referenziert werden können.

26 Vgl. Clements, Paul et al.: Documenting Software Architectures, 2008, S. 228 - 237
27 Vgl. Reilly, Annette D. et al.: IEEE Std 1063-2001, 2001, S. 5ff

3.2 Format

Damit thematische Rollen in einer Dokumentation beschrieben werden können, müssen sie darin durch ein Schlüsselwort ausgezeichnet werden können. Ein Format muss daher eine Auszeichnungsweise dafür anbieten. Es wurde auch festgestellt, dass nicht jede Rolle und somit jede Dokumentation für jeden Leser interessant ist und weiterhin unterscheiden sich die gewünschten Detailgrade der Dokumentation für unterschiedliche Leser (z. B. zwischen Entwicklern und Systemarchitekten). Das macht eine Unterteilung in adressatenspezifische Beschreibungen nötig. Eine mögliche Auswahl an Adressaten ist in Tabelle 3.1 gezeigt.

Adressat	**Beschreibung**
Entwickler	Ein Entwickler benötigt detaillierte Informationen über die Nutzungsweise und Semantik von Operationen, deren Parameter und Resultate.
Manager	Manager benötigen die Dokumentation für Planungszwecke. Sie brauchen nur eine hoch aggregierte Beschreibung der Funktionalität und Metriken um Komplexität messen zu können, um so Schätzungen für z. B. Entwicklungszeit zu treffen und auch geeignetes Personal einzuteilen.
Architekt	Ein Architekt ist an der Wiederverwendbarkeit, der Variabilität, der präzisen Semantik und den Qualitätseigenschaften von Komponenten (z. B. Schnittstellen und deren Operationen) interessiert. Weiterhin benötigt er ein Basisverständnis von der Implementierungsweise.
Administrator	Ein Administrator baut ein System aus seinen Bestandteilen zusammen und hat ein besonderes Interesse an das Zusammenspiel der Komponenten.
Tester	Ein Tester benötigt detaillierte Informationen über alle Operationen, Ressourcen und Funktionalitäten, damit er aussagekräftige und vollständige Tests entwickeln kann.

Tabelle 3.1: Auswahl an möglichen Adressaten[28]

28 Vgl. Clements, Paul et al.: Documenting Software Architectures, 2008, S. 237ff

Es sollen auch nicht nur explizite Signaturelemente einer Operation, sondern auch implizite Elemente dokumentiert und ausgezeichnet werden können. Ein explizites Element einer Signatur ist ein an der Signatur sichtbares Element, wie z. B. Ein- und Ausgabeparameter. Ein implizites Element ist demnach ein an der Signatur nicht sichtbares Element, das aber trotzdem zum Vertrag der Schnittstelle bzw. Operation gehört. Dies kann z. B. eine Formel oder eine nicht als Parameter übergebene, aber verwendete Datenquelle sein. Daraus ergibt sich, dass ein Dokumentationsblock aus einer thematischen Rolle, einem Sichtbarkeitsbereich (Scope), der implizit (IMPLICIT) oder explizit (EXPLICIT) ist, und einer Menge von individuellen Dokumentationen für verschiedene Adressaten besteht (siehe Abbildung 3.1).

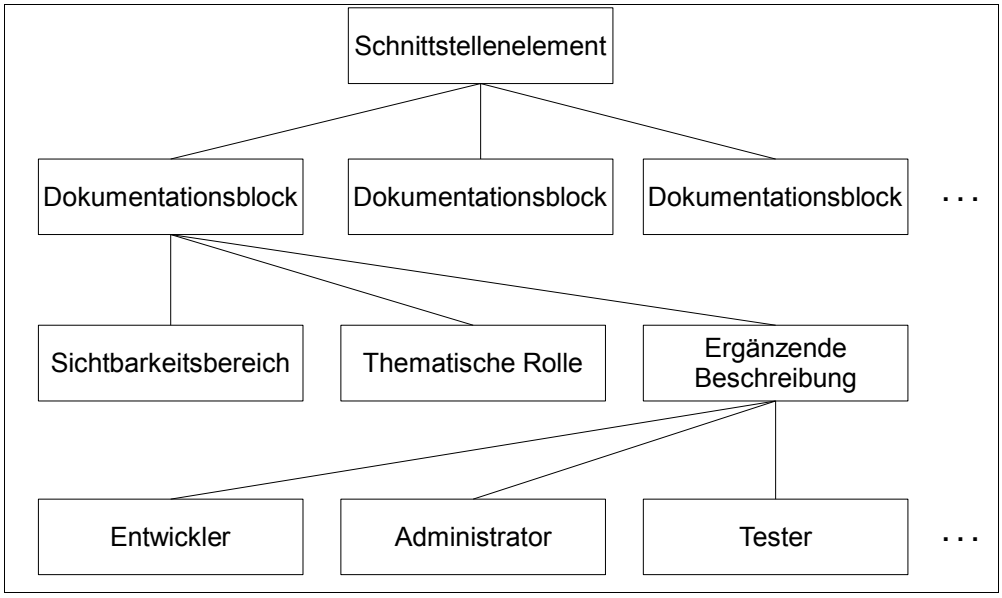


Abbildung 3.1: Aufbau eines Dokumentationsblocks

3.3 Dokumentationswerkzeug

In der Regel enthalten nur Operationsbezeichner ein Verb, bei einem Parameter z. B. ist es eher ungewöhnlich. Da thematische Rollen nur von einem Verb abgeleitet werden können, greift der Ansatz sehr wahrscheinlich nur für Operationen. Aber zur Erleichterung der Dokumentation sollen Elemente aus dem Schnittstellenvertrag, die global für die Schnittstelle gelten (z. B. Ressourcen und Konfigurationen), auch global mit einer Menge von den oben beschriebenen Dokumentationsblöcken dokumentiert werden können (z. B. an der Schnittstelle selbst). Weiterhin sollen nicht nur die sichtbaren Parameter einer Operation, also die Eingaben, Ausgaben und Fehler bzw. Ausnahmen allgemein dokumentiert werden können, sondern auch einzelne Attribute innerhalb der Parameter, die von außen erreichbar sind. Dadurch sollen thematische Rollen und Dokumentationen direkt einem betreffenden Attribut zugeordnet werden können. Die Bedeutung der Attribute wird dann am Verwendungsort (die Operation) dokumentiert. Das hat zur Folge, dass kontextabhängige Fallunterscheidungen in der Verwendung des Attributs bei der direkten Dokumentation des inneren Attributs entfallen.

Ein Beispiel soll dieses Prinzip erläutern. Dazu wird der Parameter `DetailLevel` der bereits bekannten Operation `GetFeedback` der eBay Trading API betrachtet. Durch diesen Parameter kann der abgerufene Informationsumfang eingeschränkt werden. Die Werte, die der Parameter annehmen kann, sind in Tabelle Tabelle 3.2 aufgelistet. Die Beschreibungen zu den Werten sind aus Platzgründen nicht mit aufgeführt. Die Beschreibung zu dem Wert `ReturnSummary` ist im Folgenden exemplarisch aufgeführt:

> *„Returns the summary data. For GetMyMessages, this detail level returns the same data whether or not you include MessageIDs or AlertIDs in the request. Returns up to 10 FolderID and FolderName values. Currently, this detail level is the only way to retrieve FolderID and FolderName values. See "GetMyMessages" in the eBay Web Services Guide for a code sample that demonstrates this.“*[29]

29 Vgl. o.V.: eBay Trading API – DetailLevelCodeType, o.J.

DetailLevelCodeType	
Enumerationswerte:	`ItemReturnAttributes, ItemReturnCategories, ItemReturnDescription, ReturnAll, ReturnHeaders, ReturnMessages, ReturnSummary`

Tabelle 3.2: Enumerationswerte von DetailLevelCodeType aus der eBay Trading API[30]

Schnittstellenelement		**Dokumentation**
Operation:	`GetFeedback`	
Eingabeparameter:		
`DetailLevelCodeType`	`DetailLevel`	[…] Applicable values: `ReturnAll` […] (Not all values in `DetailLevelCodeType` apply to this field.)

Tabelle 3.3: Auszug aus der Schnittstellenbeschreibung der Operation GetFeedback aus der eBay Trading API[31]

Der erste Satz der Beschreibung zu dem Wert `ReturnSummary` enthält die Bedeutung des Wertes. Die folgende Beschreibung enthält kontextspezifische Verhaltensweisen, die bei der allgemeinen Beschreibung nicht stehen sollte. In Tabelle 3.3 sieht man, dass in der Dokumentation von `GetFeedback` textuell verfasst ist, dass nur der Wert `ReturnAll` anwendbar ist. Nach dem Dokumentationsprinzip der thematischen Rollen würde z. B. die Rolle CRITERION, dem Attribut `DetailLevel` zugeordnet werden und als Beschreibung, dass nur der Wert `ReturnAll` erlaubt ist. Die zusätzliche Dokumentation in der Enumeration würde entfallen.

Mit Anwendung dieses Prinzips werden Dokumentationen dort gepflegt, wo sie Anwendung finden, und blähen die allgemeine Dokumentation eines Elements nicht unnötig auf.

30 Vgl. o.V.: eBay Trading API – DetailLevelCodeType, o.J.
31 Vgl. o.V.: eBay Trading API, o.J., „GetFeedback“

Ein Dokumentationswerkzeug sollte eine Dokumentation direkt in der Quelldatei speichern. Dadurch können inkonsistente Versionsstände und die Portierung zwischen verschiedenen Systemen (Entwicklungsumgebung bzw. Quelldatei und z. B. einem Redaktionssystem) bestmöglich vermieden werden. Hierbei ist es aber wichtig, dass die Quelldatei nur um die Dokumentation verändert wird und nicht die Semantik des Inhalts. Außerdem soll sich das Werkzeug direkt in eine Entwicklungsumgebung integrieren. Die Dokumentation von Schnittstellen lässt sich dadurch besser in den Entwicklungsprozess einbinden. Durch das ständige Wechseln zwischen verschiedenen Programmen (Entwicklungsumgebung und Redaktionssystem) würde es immer einen Bruch in dem Entwicklungsprozess geben.

Auch ein Dokumentationsautor, der sich mit Quelltexten nicht so gut auskennt, soll das Dokumentationswerkzeug nutzen können. Über eine grafische Oberfläche soll er die Schnittstellenelemente mit den oben beschriebenen Dokumentationsblöcken dokumentieren können. Für ein aus einem Operationsbezeichner extrahiertes Verb soll das Werkzeug das thematische Raster ableiten und in der Oberfläche anzeigen.

Der Dokumentationsansatz ist unabhängig von einer Programmier- oder Beschreibungssprache. Sie muss lediglich eine Art von Kommentar im Quelltext erlauben, damit die Dokumentation gespeichert werden kann und Schnittstellen mit Operationen beinhalten. Für die Dokumentation werden immer die gleichen Informationen erfasst, weswegen die grafische Oberfläche auch keine Bindung zur dokumentierenden Programmier- oder Beschreibungssprache braucht. Diese Teile sind daher prädestiniert wiederverwendet zu werden. Ein Werkzeug sollte also um weitere Programmier- und Beschreibungssprachen erweiterbar sein. Die Oberfläche und die Logik der thematischen Raster soll dabei wiederverwendet werden.

3.4 Vorgaben der AKRA GmbH

Entwicklungsschwerpunkte bei der AKRA GmbH liegen u. a. bei Java und Webservices. Als Entwicklungsumgebung wird dafür hauptsächlich Eclipse[32] eingesetzt. Daher liegt ein besonderes Interesse bei der Dokumentation von Java-Quelltexten und Schnittstellenspezifikationen von Webservices in der Webservice Description Language (WSDL), sowie die Integration eines Werkzeugs in die Entwicklungsumgebung Eclipse.

Bei der AKRA GmbH wird ausschließlich in Englisch entwickelt, daher soll sich die Verbanalyse für die thematischen Raster auf englische Verben beschränken. Die Komponente zum Ableiten der thematischen Raster (`ThematicGridService`) wird von Herrn Krause vorgegeben.

32 Siehe o.V.: Eclipse, o.J.

4 Existierende Dokumentationssysteme

Es existieren viele veröffentlichte Dokumentationssysteme. In diesem Kapitel werden eine Auswahl von 19 solcher Systeme exemplarisch betrachtet, die eigens für diese Evaluierung im Internet recherchiert wurden.

Aus den Eigenschaften der betrachteten Dokumentationssystemen werden zunächst Kategorien abgeleitet, in die die Systeme gruppiert werden können. Daraufhin werden die Systeme überprüft inwieweit sie den in Kapitel 3 aufgestellten Anforderungen genügen und zum Schluss werden die Ergebnisse der Evaluierung zusammengefasst.

4.1 Kategorien von Dokumentationssystemen

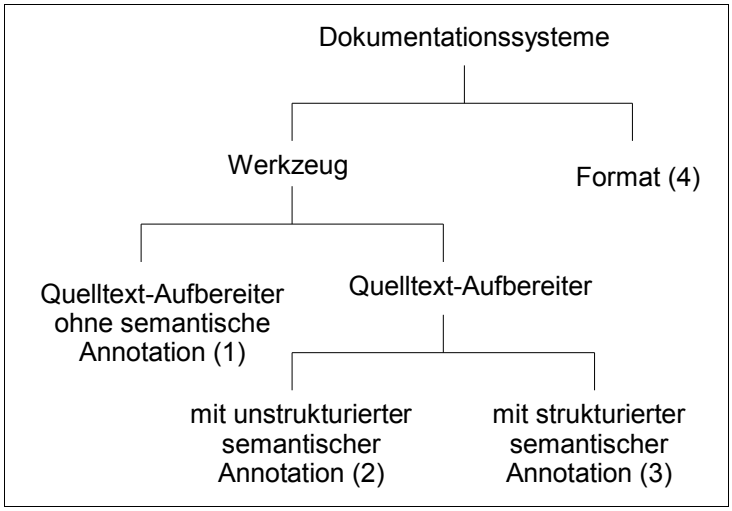


Abbildung 4.1: Kategorien von Dokumentationssystemen

Bei der Untersuchung der Dokumentationssysteme hat sich herausgestellt, dass diese viele ähnliche Eigenschaften und Funktionen haben. Von den Gemeinsamkeiten ausgehend, konnten Kategorien abgeleitet werden, in denen die Dokumentationssysteme eingeteilt werden können (siehe Abbildung 4.1).

Kategorie 1: Quelltext-Aufbereiter ohne semantische Annotation

In der Kategorie 1 befinden sich Werkzeuge, die sich nur auf den Quelltext konzentrieren und diesen analysieren, wie z. B. Help Generator und Universal Report. Sie bereiten den Quelltext auf, indem z. B. die Eingaben, Ausgaben und Fehler einer Operation gegliedert dargestellt werden oder alle Stellen referenziert werden, von wo aus eine Operation aufgerufen wird. Hierbei werden keine Informationen aus Kommentaren oder anderen Quellen berücksichtigt. Die Dokumentation ist hier das Ergebnis dieser Aufbereitung, das in ein anderes Format, meist HTML, gespeichert wird.

Kategorie 2: Quelltext-Aufbereiter mit unstrukturierter semantischer Annotation

Die Werkzeuge der Kategorie 2 (z. B. RDoc) bereiten den Quelltext auch auf, nehmen aber noch unstrukturierte Informationen aus Quelltextkommentaren auf. Mit unstrukturierten Informationen ist einfacher Freitext als Beschreibung gemeint, z. B. einer Operation.

Kategorie 3: Quelltext-Aufbereiter mit strukturierter semantischer Annotation

In der Kategorie 3 wird ein durch Schlüsselwörter strukturierter Text, der in Quelltextkommentaren gespeichert ist, mit in die Dokumentation aufgenommen. Die Werkzeuge aus dieser Kategorie unterstützen meistens ein eigenes Format und teilweise auch mehrere verschiedene Formate. Hierbei handelt es sich um Formate, die Schlüsselwörter wie bei Javadoc (z. B. „@param“, „@return“, „@see“) oder natürlich sprachliche wie bei Natural Docs (z. B. „Parameters:“, „Returns:“ oder „See Also:“), verwenden, aber auch Formate, die XML Elemente in die Kommentare integrieren, um die Beschreibungen zu strukturieren.

Bei den Kategorien 1 bis 3 steht mehr das Werkzeug im Vordergrund, das den Quelltext ggf. zusammen mit den Inhalten aus Kommentaren aufbereitet und in ein anderes Format, hauptsächlich HTML, exportieren kann. Die Dokumentationen werden i. d. R. in Kommentaren direkt im Quelltext gespeichert.

Kategorie 4: Format

Die Kategorie 4 enthält Formate, die u. a. zur Schnittstellendokumentation verwendet werden können. Bei der Nutzung dieser Formate wird die Dokumentation außerhalb des Quelltextes gespeichert. Es findet keine Aufbereitung des vorhandenen Quelltextes statt. DITA (Darwin Information Typing Architecture) und DocBook sind z. B. solche auf XML basierenden Formate, in denen technische Dokumentationen verfasst werden können. Diese Formate können durch zusätzliche Programme in andere Formate (z. B. HTML oder PDF) exportiert werden. Zum Erstellen von Dokumentationen in diesem Format reicht ein einfacher Text- bzw. XML-Editor.

4.2 Unterstützung der Dokumentation mit thematischen Rollen

Da die Dokumentationssysteme der Kategorie 1 keine zusätzlichen Dokumentation berücksichtigen und der Kategorie 2 nur Freitext erlauben, genügen sie den Anforderungen von Grund auf nicht. Nur ein durch Schlüsselwörter strukturiertes Format, wie es in der Kategorie 3 und 4 verwendet wird, genügt den Anforderungen u. a. thematische Rollen und adressatenspezifische Beschreibungen zu unterstützen.

Wie in den Tabellen 4.1 und 4.2 zu erkennen ist, sind bei den meisten Formaten fast alle der von Clements et al. empfohlenen Beschreibungen und auch weitere Metadaten mit einem Schlüsselwort vorgesehen, wie z. B. der Autor einer Schnittstelle oder ob die Schnittstelle veraltet ist. Die meisten sind auch um weitere Schlüsselwörter erweiterbar, weswegen teilweise fehlende Schlüsselwörter individuell nachgepflegt werden können. Die Anforderungen von Clements et al. an eine Schnittstellendokumentation scheinen auch in der Praxis verbreitet erkannt und angewendet zu werden.

Name	Unterstützte Dokumentationstypisierungen[33]						
	Eingabe	**Ausgabe**	**Fehler**	**Autor**	**Version**	**Seit welcher Version enthalten**	**Veraltet**
CppDoc	+	+	+	+	+	+	+
Doc-O-Matic	+	+	+	+	+	+	+
DOC++	+	+	+	+	+	+	+
Document! X	+	+	+	o	o	o	o
Doxygen	+	+	+	+	+	+	+
GenHelp	+	+	+	+	+	+	+
HeaderDoc	+	+	+	+	+	+	+
Javadoc	+	+	+	+	+	+	+
Natural Docs	+	+	+	+	+	o	+
ROBODoc	+	+	+	+	o	o	o
TwinText	+	+	+	+	+	-	+
VSdocman	+	+	+	+	+	+	o
YARD	+	+	+	+	+	+	+
DITA	o	o	o	o	o	o	o
DocBook	o	o	o	o	o	o	o

Tabelle 4.1: Unterstützte Dokumentationstypisierungen von den untersuchten Kategorie 3 und 4 Dokumentationssystemen

Für nicht sichtbare Elemente gibt es zwei Kriterien, die geprüft werden (siehe Tabelle 4.3). Zum einen ob diese abbildbar sind, d. h. ob Schlüsselwörter zur Beschreibung dieser Elemente vorgesehen sind, die nicht direkt an der Schnittstelle sichtbar sind (aber Vertragsbestandteil sind). Bei Operationen fallen darunter z. B. implizit verwendete Ressourcen, Seiteneffekte und Attribute aus den inneren Strukturen von Parametern. Nur zwei von den 19 Referenzsystemen sehen so eine Beschreibung von sich aus vor. HeaderDoc bietet z. B. das Schlüsselwort `@dependency` zur Beschreibung von Ressourcen, von denen die Schnittstelle abhängt, und ROBODoc bietet das Schlüsselwort `SIDE EFFECTS` an, um Seiteneffekte zu beschreiben. Das ist aber jeweils nur ein von mehreren nicht sichtbaren Elementen, die beschrieben werden sollen. Aber alle, um weitere

33 Legende: +: ist vorgesehen; -: nicht möglich; o: nicht vorgesehen, aber möglich

Schlüsselwörter erweiterbaren Formate, können dahingehend angepasst werden, nicht sichtbare Elemente zu unterstützen. Das gilt auch für die adressatenspezifischen Beschreibungen, die kein Dokumentationssystem vorsieht.

Name	Unterstützte Dokumentationstypisierungen				
	Referenz	**Beschreibung**	**Untypisierter Bereich (Freitext)**	**Zielgruppen-gerichtete Beschreibung**	**Erweiterbar um Typen**
CppDoc	+	-	+	-	-
Doc-O-Matic	+	+	+	o	+
DOC++	+	+	+	-	-
Document! X	+	+	-	o	+
Doxygen	+	+	+	o	+
GenHelp	+	+	+	o	+
HeaderDoc	+	+	+	o	+
Javadoc	+	o	+	o	+
Natural Docs	+	+	+	o	+
ROBODoc	+	+	-	o	+
TwinText	+	+	-	-	-
VSdocman	+	+	+	o	+
YARD	+	+	+	o	+
DITA	o	o	o	o	o
DocBook	o	o	o	o	o

Tabelle 4.2: Fortsetzung: Unterstützte Dokumentationstypisierungen von den untersuchten Kategorie 3 und 4 Dokumentationssystemen

Zum anderen wird für nicht sichtbare Elemente geprüft ob diese ableitbar sind. Wie in Tabelle 4.3 zu erkennen ist, unterstützt keines der untersuchten Dokumentationssysteme den Autor durch eine semantische Analyse, um diese nicht sichtbaren, aber beschreibungswürdigen Elemente herauszufinden. Einige bieten die Erstellung eines textuellen Rasters von den sichtbaren Elementen einer Operation im Quelltext an und einige andere, wie z. B. VSdocman und Doc-O-Matic über eine grafische Oberfläche. Die Formate aus der Kategorie 4 offerieren nichts dergleichen.

Sie haben lediglich eine eigene Gliederungsvorschrift, nach der Dokumentationen strukturiert werden können.

Name	Nicht sichtbare Elemente	
	abbildbar	**ableitbar**
CppDoc	o	-
Doc-O-Matic	o	-
DOC++	-	-
Document! X	o	-
Doxygen	o	-
GenHelp	o	-
HeaderDoc	+	-
Javadoc	o	-
Natural Docs	o	-
ROBODoc	+	-
TwinText	-	-
VSdocman	o	-
YARD	o	-
DITA	o	-
DocBook	o	-

Tabelle 4.3: Unterstützung nicht sichtbarer Elemente von den untersuchten Kategorie 3 und 4 Dokumentationssystemen

Die Werkzeuge wurden auch auf die Erweiterbarkeit von unterstützten Programmier- und Beschreibungssprachen geprüft. Die Formate (Kategorie 4) sind hierbei außen vor, da sie unabhängig von einer Programmiersprache eingesetzt werden. Nur fünf der 17 untersuchten Werkzeuge sind erweiterbar um Sprachen, die Kommentare erlauben. Die anderen haben ihren festen und geringeren Satz an unterstützten Programmier- und Beschreibungssprachen. Diese Korrelation zwischen der Anzahl an unterstützten Sprachen und der Eigenschaft, ob das Werkzeug erweiterbar konzipiert wurde, sieht man gut in der Tabelle 4.4.

Name	Anzahl unterstützter Programmier- und Beschreibungssprachen	Erweiterbar für Sprachen mit Kommentaren
CppDoc	1	Nein
Javadoc	1	Nein
RDoc	1	Nein
YARD	1	Nein
Imagix 4D	2	Nein
VSdocman	2	Nein
DOC++	3	Nein
GenHelp	3	Nein
Document! X	4	Nein
Help Generator	4	Nein
HeaderDoc	9	Nein
Doxygen	11	Nein
Doc-O-Matic	11	Ja
ROBODoc	15	Ja
Natural Docs	19	Ja
TwinText	25	Ja
Universal Report	25	Ja

Tabelle 4.4: Korrelation zwischen Erweiterbarkeit und Anzahl unterstützter Programmier- und Beschreibungssprachen von den untersuchten Dokumentationssystemen

Die in diesem Kapitel gezeigten Tabellen sind sind nur auszugsweise dargestellt. Die vollständigen Tabellen befinden sich im Anhang A.

4.3 Fazit

Aus den exemplarisch betrachteten Dokumentationssystemen wird erkennbar, dass die existierenden Werkzeuge hauptsächlich bei der Erstellung einer Dokumentation aus dokumentierten Quelltexten gute Unterstützung leisten. Sie bereiten diese Informationen gut auf und Überführen sie in eine für Menschen gut lesbare Form (HTML, PDF u. ä.). Sie sind dabei aber sehr eng an die Möglichkeiten des verwendeten Formats gebunden. Die Formate aus der Kategorie 4 sind in ihrer Verwendung flexibler, haben aber nicht unbedingt einen direkten Bezug zum Quelltext. Bei der wichtigsten untersuchten Funktion, der Unterstützung beim Dokumentieren mit z. B. einer semantischen Analyse, wie es der Ansatz der thematischen Raster macht, fallen alle untersuchten Dokumentationssysteme durch. Das Hauptaugenmerk liegt bei den sichtbaren Elementen einer Schnittstelle, die zur Dokumentation angeboten werden. Thematische Rollen, die nicht sichtbare Elemente aufzeigen können, sind zur Zeit nur durch kreative Leistung des Autors erfassbar. Weiterhin sind thematische Rollen, adressatenspezifische Beschreibungen und implizite Elemente einer Schnittstelle nur teilweise abbildbar.

Damit ein Autor von Schnittstellendokumentationen und speziell O-Doku technische Unterstützung beim Dokumentieren erhält, soll das Werkzeug iDocIt! entwickelt werden, das das Dokumentieren mit thematischen Rastern erlaubt. Dadurch soll der kreative Aufwand des Autors und der Dokumentationsumfang reduziert, sowie inhaltliche Lücken in der O-Doku aufgezeigt und so vermieden werden. Das Werkzeug soll auch die Erfassung adressatenspezifischer Beschreibungen und die Dokumentation von impliziten Elementen einer Schnittstelle unterstützen.

5 Systementwurf

Wie sich im vorherigen Kapitel herausgestellt hat, erfüllt keines der untersuchten Dokumentationssysteme das wichtigste Kriterium, nämlich die Unterstützung beim Dokumentieren durch die Vorgabe von nicht sichtbaren Vertragselementen einer Schnittstelle. Andere Anforderungen werden nur teilweise erfüllt. Daher wird in diesem Kapitel das neue Dokumentationswerkzeug iDocIt! entworfen.

Bei der Anforderungsanalyse wurde festgestellt, dass der Ansatz der thematischen Raster unabhängig von einer Programmier- oder Beschreibungssprache ist. Dies gilt ebenso für den strukturellen Aufbau der Dokumentationsblöcke. Daher soll das Werkzeug in fixe und variable Komponenten aufgeteilt werden, die über Schnittstellen miteinander kommunizieren (siehe Abbildung 5.1).

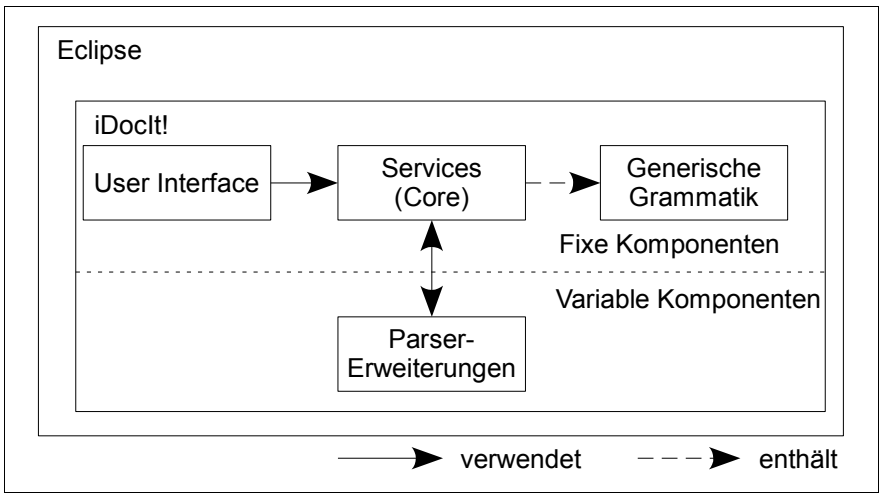


Abbildung 5.1: Aufteilung in fixe und variable Komponenten von iDocIt! im Kontext von Eclipse

Die fixen Komponenten von iDocIt! sind die grafische Oberfläche (User Interface) und die Plattform (Core) mit ihren Diensten, die sie zur Verfügung stellt. Die Parser-Erweiterungen sind die variablen Komponenten. Sie können sich variabel an der Plattform an- und abmelden.

Im Folgenden wird zunächst die generische Grammatik entwickelt, die die meisten Schnittstellenstrukturen abbilden soll. Danach werden die Komponenten von iDocIt!

und deren Modularisierung als Eclipse Plugins vorgestellt. Die Parser-Erweiterungen und die grafische Oberfläche werden danach detaillierter erläutert. Abschließend wird beschrieben, wie die einzelnen Komponenten über die bereitgestellten Services miteinander agieren.

5.1 Grammatiken

In diesem Kapitel wird die erwähnte generische Grammatik zur Abbildung der meisten Programmier- und Beschreibungssprachen entwickelt. Das Konzept der formalen Grammatiken wird in einem hohen Abstraktionsniveau verwendet, dazu wird zuerst eine kurze Einführung in die formalen Grammatiken gegeben.

5.1.1 Grundlagen formaler Grammatiken

In der theoretischen Informatik ist eine Grammatik G ein Regelwerk, wie syntaktisch korrekte Wörter einer formalen Sprache L gebildet werden. Sie ist als ein Viertupel $G = (N, \Sigma, P, S)$ definiert. Das Vokabular einer Grammatik besteht aus einer endlichen Menge von Variablen, sogenannten Nichtterminalsymbolen (oder Nichtterminale) N und einer endlichen Menge von Zeichen (Terminalsymbole oder Terminale), die das Alphabet Σ bilden. Terminale können z. B. Ziffern, Buchstaben und Sonderzeichen sein. Die Nichtterminalsymbole sind nur in Zwischenschritten einer Ableitung enthalten und dürfen nicht im endgültigen Wort vorkommen. Ausgehend von einem Startsymbol $S \in N$ lassen sich durch Produktionsregeln aus einer Menge von Regeln P Wörter ableiten. Ein Wort kann dabei aus keinem (leeres Wort ε), einem oder mehreren Terminalen aus dem definierten Alphabet Σ bestehen. Die Menge aller Wörter, die aus Σ abgeleitet werden können, wird mit Σ^* bezeichnet.

Mit einer formalen Grammatik können also Wörter gebildet werden, die zu einer bestimmten formalen Sprache gehören, aber es kann auch überprüft werden, ob ein Wort zu einer bestimmten Sprache gehört (Wortproblem).[34,35]

34 Vgl. Socher, Rolf: Theoretische Grundlagen der Informatik, 2008, „Alphabet“, S. 16f
35 Vgl. Socher, Rolf: Theoretische Grundlagen der Informatik, 2008, „Grammatik“, S. 65ff

Grammatik G = (N, Σ, P, S) mit
N = { A, B }
Σ = { a, b }
P = { A → aA | B, B → bB }
S = A ∈ N.

Tabelle 5.1: Beispiel einer Grammatik G

Eine beispielhafte Grammatik *G* ist in Tabelle 5.1 definiert. Die Nichtterminale werden hier als Großbuchstaben dargestellt und die Terminale als Kleinbuchstaben. Der senkrechte Strich „|“ in der einen Produktionsregel bedeutet, dass entweder das eine oder das andere Ergebnis aus der Produktion resultieren kann. Dadurch können mehrere Produktionsregeln zusammengefasst werden. Bei der längeren Schreibweise werden die Regeln alle einzeln aufgeführt: $A \rightarrow aA \mid B \;\Leftrightarrow\; A \rightarrow aA, A \rightarrow B$.

Ausgehend vom Startsymbol *A* können Wörter produziert werden, die mit einer beliebigen Anzahl *a*'s anfangen (auch die leere Menge, also kein *a*, ist erlaubt) und mit mindestens einem bis beliebig vielen *b*'s aufhört, z. B.:

$$A \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aabB \rightarrow aabbB \rightarrow aabbb \in \Sigma^*.$$

Die Produktionsregeln werden so lange angewendet, bis alle Nichtterminale durch Terminale ersetzt worden sind, da ein Wort nur aus Terminalen bestehen darf. Es wird die formale Sprache $L(G) = \{a^n b^m \mid n, m \in \mathbb{N} \wedge n \geq 0 \wedge m \geq 1\}$ gebildet.

Eine Grammatik heißt kontextfrei, wenn sie ausschließlich Produktionsregeln besitzt, bei denen auf der linken Seite der Produktionsregeln genau ein Nichtterminalsymbol steht, z. B. $A \rightarrow aA$, mit $A \in N$. Eine Grammatik ist kontextabhängig, wenn eine Ableitung eines Nichtterminals nur in einem bestimmten Kontext möglich ist, z. B. $xAy \rightarrow xaAy$, mit $A \in N$. Das bedeutet, dass die Regel nur angewendet werden darf, wenn *A* zwischen *x* und *y* steht, also in einem bestimmten Kontext.[36]

36 Vgl. Socher, Rolf: Theoretische Grundlagen der Informatik, 2008, „Kontextfreie Grammatiken“, S. 74ff

5.1.2 Generische Grammatik für Schnittstellen

In diesem Kapitel wird eine kontextfreie Grammatik entwickelt, die die Struktur der meisten Programmier- und Beschreibungssprachen abbilden kann. Aus den Vorgaben der AKRA GmbH (siehe Kapitel 3.4) resultiert für diese Bachelorarbeit, dass Parser für Java und WSDL entwickelt werden. Daher orientiert sich die zu entwickelnde Grammatik hauptsächlich an diesen Sprachen. Für Java wurde die Sprachspezifikation in der aktuellen dritten Edition[37,38] herangezogen und für WSDL die Version 1.1[39,40], da die Version 2.0[41] bei der AKRA GmbH noch nicht eingesetzt wird.

Ausgehend davon, dass eine Schnittstelle aus irgendeiner Quelle ausgelesen werden kann, wird diese i. d. R. eine Datei sein, auf die sich auch iDocIt! beschränkt. Die Quelle soll in der allgemeinen Schnittstellenstruktur aber trotzdem allgemein `InterfaceArtifact` heißen. In diesem Artefakt können mehrere Schnittstellendefinitionen enthalten sein, in Java z. B. mehrere Schnittstellen (`interfaces`). Selbiges gilt in der WSDL. Hier kann ein Artefakt mehrere `<portType>`-Elemente enthalten, die zur Beschreibung einer Schnittstelle verwendet werden. Abstrahiert man die erwähnten Teile zu einer allgemeinen Struktur, besteht ein Artefakt aus einer Menge von allgemeinen Schnittstellenstrukturen, die `Interface` heißen sollen.

Eine Schnittstelle in Java enthält eine Menge von Operationen (Methoden) und kann aber auch wieder eine Menge innerer Schnittstellen enthalten. Bei der WSDL enthält das `<portType>`-Element nur Operationen (`<operation>`-Elemente) (siehe Tabelle 5.2). Demnach muss eine allgemeine Schnittstellenstruktur aus einer Menge von Operationen, die `Operation` heißen sollen, und einer Menge von inneren Schnittstellen (`Interface`) bestehen.

37 Vgl. Gosling, James et al.: Java Language Specification, 2005
38 Siehe Anhang C Java Interface Deklaration
39 Vgl. Christensen, Erik et al.: WSDL 1.1, 2001
40 Siehe Anhang B WSDL 1.1 Spezifikation
41 Vgl. Chinnici, Roberto et al.: WSDL Version 2.0, 2007

```
<wsdl:portType name="nmtoken"> *
  <wsdl:operation name="nmtoken"> *
    <wsdl:input name="nmtoken"? message="qname" /> ?
    <wsdl:output name="nmtoken"? message="qname" /> ?
    <wsdl:fault name="nmtoken" message="qname" /> *
  </wsdl:operation>
</wsdl:portType>
```

Tabelle 5.2: Auszug der WSDL 1.1 Spezifikation für das <portType>-Element[42,43]

In Java besteht eine Operation aus beliebig vielen Eingabeparametern, einem Ergebnistypen und einer Menge möglicherweise auftretender Ausnahmen. In der WSDL ist es ein wenig komplizierter. Eine Operation hat ein oder kein <input>- und <output>-Element für die Eingaben und Ausgaben, sowie eine beliebige Menge von <fault>-Elementen für eventuelle Ausnahmen (siehe Tabelle 5.2). Jedes dieser Elemente referenziert ein <message>-Element. Meist wird für jede Operation ein Input-Message-Typ und ein Output-Message-Typ definiert. Es ist aber auch möglich., dass sich mehrere Operationen einen Message-Typ teilen. Ein <message>-Element besteht aus beliebig vielen <part>-Elementen, die entweder ein Element referenzieren oder einen komplexen oder einfachen / primitiven Typen[44] haben (z. B. ein Ganzzahltyp) (siehe Abbildung 5.2).

42 Legende: ?: 0 bis 1 Mal vorhanden; *: 0 bis beliebig oft vorhanden
43 Vgl. Anhang B
44 Vgl. Thompson, Henry S. et al.: XML Schema Part 1, 2001, „Complex Type Definitions“ und „Simple Type Definitions“

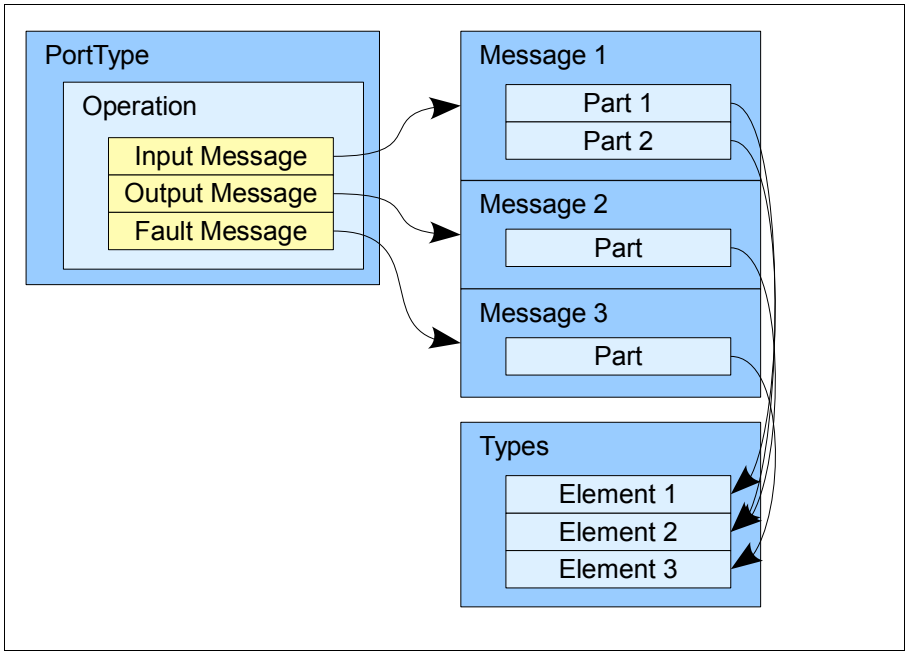


Abbildung 5.2: Beispielhafter Aufbau eines `<portType>`-Elements der WSDL 1.1 Spezifikation

Für Java würde es reichen, wenn für die Eingaben und Ausnahmen Mengen von Parametern und für den Ergebnistypen ein einzelner Parameter verwendet werden würde, WSDL benötigt an dieser Stelle aber mehr. Eine Operation in WSDL verwendet für die Ein- (`<input>`) und Ausgaben (`<output>`) jeweils ein `<message>`-Element und für Fehler (`<fault>`) eine Menge von `<message>`-Elementen. Jedes `<message>`-Element besteht jeweils wiederum aus einer Menge von `<part>`-Elementen. Weiterhin besitzt ein `<message>`-Element selbst Eigenschaften, weswegen es sich nicht als Menge von Parametern (`<part>`-Elemente) darstellen lässt. Es muss ein Element in der allgemeinen Struktur definiert werden, das ein `<message>`-Element abbildet. Dieses Element soll `Parameters` heißen, da es eine Menge von Parametern zusammenfasst. Für eine Methode in Java wird ein `Parameters` Element die Menge der Eingabeparameter enthalten und ein weiteres `Parameters` Element eine einelementige Menge für den Ergebnistypen. Da WSDL eine Menge der `Parameters` Elemente für die Fehler benötigt, wird in Java

eine Menge von `Parameters` mit jeweils einer Ausnahme gebildet. Eine einzelne Java-Ausnahme wird also wie eine WSDL-`<message>` behandelt.

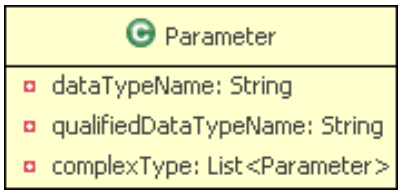


Abbildung 5.3: Die Klasse `Parameter` mit den zusammengefassten Attributen

Wie in den Anforderungen beschrieben, sollen auch Attribute der Eingaben, Ausgaben und Fehler bzw. Ausnahmen dokumentiert werden können. Da alle prinzipiell die gleichen Eigenschaften (Attribute) haben, sollen sie unter dem Begriff `Parameter` zusammengefasst werden (siehe Abbildung 5.3).

Um die Attribute eines Parameters abzubilden, erhält die Klasse eine weitere Struktur. In Java kann ein Parameter ein strukturierter (z. B. `List`, `Set`, usw.) oder ein unstrukturierter Datentyp (z. B. `int`, `long`, `double`, `char`, usw.) sein. In WSDL beschreibt ein komplexer oder einfacher Typ die Struktur eines Elements. Das Element selbst braucht demnach nicht separat behandelt werden, da die Typen ausschlaggebend sind. Bei einem einfachen Typen (`SimpleType`) braucht ein `Parameter` keine weitere Struktur, da es keine weiteren Attribute gibt. Bei einem komplexen Typ bzw. strukturierten Datentyp können beliebig viele Attribute vorhanden sein. Da Attribute und Parameter die gleichen Eigenschaften haben, werden sie auch als `Parameter` abgebildet. Ein `Parameter` erhält somit für die inneren Attribute eine Menge von `Parameter` (`ComplexType`). Damit ist auch gleich abgedeckt, dass die Attribute selbst wieder einfache oder komplexe bzw. strukturierte Typen sein können.

Die meisten Elemente haben neben den strukturellen Komponenten auch weitere Attribute, die mit dem Nichtterminal `Atttributes` exemplarisch dargestellt werden

sollen. Die darin beschriebenen Elemente sind nicht vollständig, enthalten aber u. a. einen Dokumentationsblock (`Documentation`).

In Tabelle 5.3 ist die allgemeine Grammatik einer Schnittstellenstruktur in EBNF (Erweiterte Backus-Naur-Form) dargestellt. Die Produktionsregeln definieren hauptsächlich die abstrakte Struktur. Attribute, die zusätzlich für die Implementierung benötigt werden, sind nicht enthalten (z. B. die Attribute `id` und `parent`). Sie werden zusammen mit der folgenden Klassenstruktur der allgemeinen Grammatik beschrieben.

Die in den Produktionsregeln der Grammatik groß geschriebenen Wörter stellen Nichtterminalsymbole dar (z. B. `Interface` und `Operation`). Terminalsymbole sind in der vereinfachten Darstellung nur für den Scope enthalten, die klein geschriebenen Wörter `explicit` und `implicit`. Weitere Terminale würden aus den Nichtterminalen `Identifier`, `String` und `SimpleType` resultieren. Letzter kann wegen der Abhängigkeit zur verwendeten Programmier- oder Beschreibungssprache nicht weiter spezifiziert werden (möglich wären z. B. `long`, `int`, `xsd:integer`, `xsd:boolean`). Die geschweiften Klammern bedeuten, dass die darin enthaltenen Elemente null bis beliebig oft wiederholt werden, also eine Menge bilden. Dies kann auch die leere Menge sein. Wenn mehrere Elemente hintereinander aufgeführt sind, definieren sie eine Folge. Sie werden alle aus dem linken Nichtterminal abgeleitet, soweit keine weiteren Optionen vorhanden sind. Ein vertikaler Strich „|“ definiert alternative Resultate der Produktionsregel. Die runden Klammern bilden eine Gruppe von Symbolen. Alternativen innerhalb der Gruppe beziehen sich ausschließlich auf die Symbole innerhalb der runden Klammern.

```
InterfaceArtifact = { Interface } Attributes
Interface         = { Operation } { Interface } Attributes
Operation         = InputParameter OutputParameter
                    { Exception } Attributes
InputParameter    = Parameters
OutputParameter   = Parameters
Exception         = Parameters
Parameters        = { Parameter } Attributes
Parameter         = ( ComplexType | SimpleType ) Attributes
ComplexType       = { Parameter }
Attributes        = Identifier Documentations
Documentations    = { Documentation }
Documentation     = Role Scope { Addressee_Doc }
Role              = String
Scope             = explicit | implicit
Addressee_Doc     = Addressee String
Addressee         = Name
Name              = String
```

Tabelle 5.3: Allgemeine Grammatik einer Schnittstellenstruktur in EBNF (vereinfachte Darstellung)

Aus der allgemeinen Grammatik kann dann eine Klassenstruktur abgeleitet werden, die in Abbildung 5.4 zu sehen ist. Da alle Elemente aus der allgemeinen Grammatik viele gleiche Attribute haben, sind diese Gemeinsamkeiten in der Klasse `SignatureElement` zusammengefasst (siehe Abbildung 5.5).

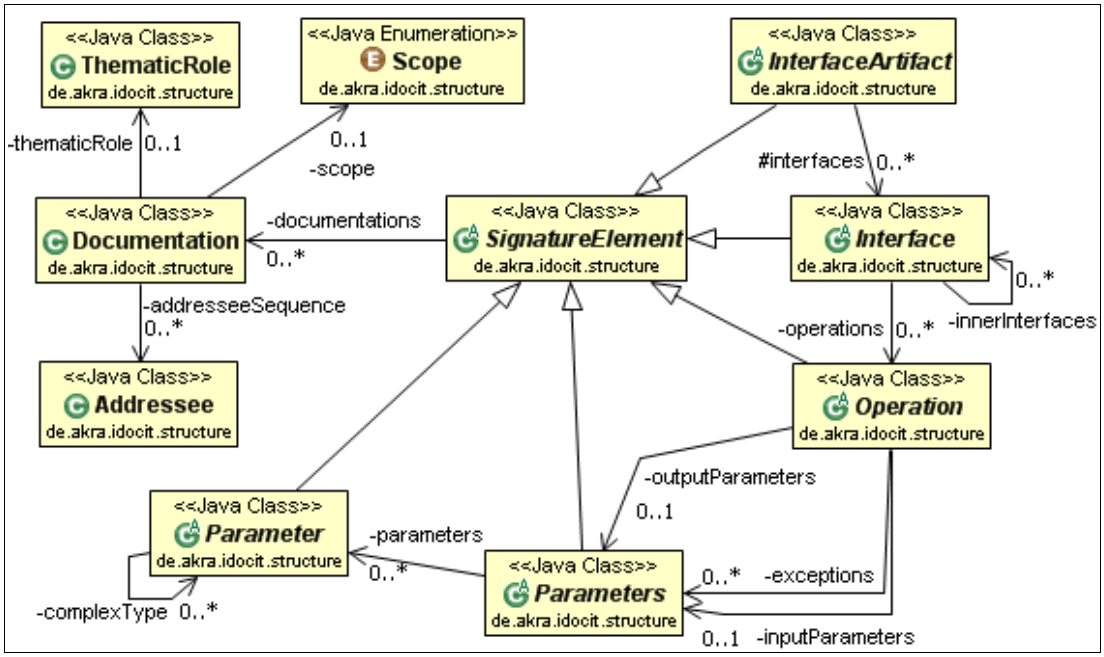


Abbildung 5.4: Klassendiagramm der allgemeinen Grammatik einer Schnittstellenstruktur

Ein `SignatureElement` besitzt eine Menge von Dokumentationsblöcken (`documentations`), wobei ein Dokumentationsblock durch die Klasse `Documentation` abgebildet ist. Eine `Documentation` Klasse enthält eine Menge von adressatenspezifischen Dokumentationen, eine thematische Rolle (`ThematicRole`) und einen Sichtbarkeitsbereich (`Scope`), der implizit oder explizit sein kann. Explizit soll für sichtbare Schnittstellenelemente verwendet werden und implizit für nicht sichtbare Elemente. In dem Beispiel aus Kapitel 2 sind die gefundenen impliziten Elemente z. B. eine Datenquelle und eine Formel.

Als weitere Attribute hat es einen einfachen Bezeichner (`identifier`), einen qualifizierten Bezeichner (`qualifiedIdentifier`) (mit Zusatz des Namensraum), eine Kategorie (`category`) (z. B. Operation oder Artifact), ein Elternelement (`parent`), ein Flag, ob eine Dokumentation dieses Elements erlaubt ist (`documentationAllowed`), und einer innerhalb einer `InterfaceArtifact` Struktur eindeutigen ID (`id`). Das statische Klassenattribut `EMPTY_SIGNATURE_ELEMENT` ist eine „leere“ Instanz des `SignatureElement`'s,

das u. a. als Wurzel- bzw. Elternelement eines `InterfaceArtifact`'s dient. Das statische Klassenattribut `ID_COUNTER` ist der Zähler der eindeutigen `id`.

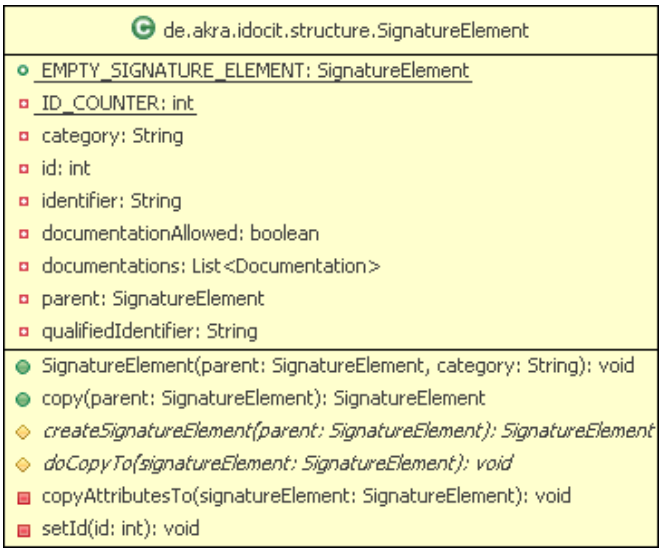


Abbildung 5.5: Auszug der abstrakten Klasse `SignatureElement`

Die ID wird benötigt, weil ein `SignatureElement` allein durch seine Attribute nicht eindeutig in einem `InterfaceArtifact` ist. Dieses Kriterium ist aber beispielsweise für die Verwaltung von Informationen zu einzelnen `SignatureElements` außerhalb der Struktur zwingend erforderlich. Es können z. B. in einem `InterfaceArtifact` zwei verschiedene `Interfaces` mit gleichen `Operations` oder in verschiedenen `Operations` Parameter mit gleichem Typ vorhanden sein. In Kapitel 6.1.1 wird darauf genauer eingegangen.

Die Klassen der Schnittstellenstruktur (`InterfaceArtifact`, `Interface`, `Operation`, `Parameters` und `Parameter`), die von `SignatureElement` erben, sind alle wiederum auch als abstrakt deklariert. Dadurch ist ein Entwickler, der einen Parser für das Werkzeug implementieren möchte, gezwungen diese Struktur zu beerben. Das hat den Vorteil, dass durch abstrakte Methoden dem Entwickler deutlicher gezeigt wird, welche Teile der Anwendungssituation entsprechend implementiert werden müssen. Ein Beispiel dazu ist die Erstellung einer tiefen Kopie eines `SignatureElements`, das in Kapitel 6.1.1 beschrieben wird.

Der Entwickler ist auch dafür verantwortlich alle nötigen Informationen in den jeweiligen Objekten zu speichern, sodass die Struktur nach dem Erstellen und Manipulieren auch wieder in die Datei bzw. Quelle zurückgeschrieben werden kann. Um das zu erreichen, müssen die einzelnen Klassen der allgemeinen Schnittstellenstruktur beerbt werden. Für die Parser-Erweiterungen für WSDL und Java ist dieses in Kapitel 6.2.3 und 6.3.3 gezeigt.

5.2 Modularisierung und Erweiterbarkeit

Wie in Kapitel 3.4 beschrieben, wird bei der AKRA GmbH hauptsächlich die Entwicklungsumgebung Eclipse eingesetzt, in die das Werkzeug iDocIt! integriert werden soll. Zur Modularisierung sollen daher die Komponenten UI, Core und die Parser-Erweiterungen (siehe Abbildung 5.1) jeweils als ein Eclipse Plugin realisiert werden.

Durch die Modularisierung soll die Funktionalität von iDocIt! auf einzelne Komponenten aufgeteilt werden. Die Komponenten sollen über definierte Schnittstellen miteinander kommunizieren. Die UI Komponente greift dabei auf die Core Komponente zu, aber nicht umgekehrt. Die Core Komponente und die Parser-Erweiterungen (z. B. die für Java und WSDL) verwenden sich gegenseitig.

Diese strikte Kapselung von Funktionalität soll die Wartbarkeit erhöhen und die späteren Weiterentwicklungen fördern. Realisiert werden soll sie mit der Zugriffsrechteverwaltung für Bundles bzw. Plugins der OSGi Spezifikation, auf der Eclipse basiert (siehe Kapitel 5.2.1).

Wenn iDocIt! als Open Source-Projekt veröffentlicht ist, sollen Entwickler unabhängig voneinander z. B. neue Parser-Erweiterungen entwickeln können. Da soll die vorgegebene Programmierkonvention der Funktionskapselung in den Komponenten dazu beitragen, dass nicht gegenseitig auf Implementierungsdetails zugegriffen wird. Beispielsweise soll die Core Komponente oder die Parser-Erweiterungen nicht auf grafische Kontrollelemente des UIs zugreifen.

Da die Komponenten als Eclipse Plugins realisiert werden sollen, werden diese zuerst allgemein beschrieben. Daraus folgend wird die Modularisierung von den iDocIt! Komponenten erläutert und beschrieben wie die Parser-Erweiterungen sich an der Plattform (Core) registrieren.

5.2.1 Eclipse Plugins

Seit der Version 3.0 basiert Eclipse auf der OSGi[45] Kernspezifikation, die in einem eigenen Eclipse-Projekt mit dem Namen Equinox zusammen mit einigen zusätzlichen Features (z. B. die Extension Registry) entwickelt wird. Bevor entschieden wurde, dass Eclipse in der Version 3.0 das OSGi Framework nutzen wird, war die Idee Eclipse auf eine Plugin-Struktur umzustellen. Der Name „Eclipse Plugin“ ist bis heute bestehen geblieben, gemeint ist aber ein sogenanntes OSGi Bundle.

Die OSGi Alliance spezifiziert eine hardwareunabhängige, dynamische Softwareplattform, die es per Komponentenmodell ermöglicht Anwendungen und ihre Dienste (Bundles und Services) zu modularisieren und zu verwalten. Einen großen Vorteil bietet die Möglichkeit Bundles, während der Programmlaufzeit hinzuzufügen, zu starten, zu stoppen, zu aktualisieren und zu entfernen. Dadurch ist ein dynamischer Lebenszyklus von Bundles möglich. Es gehört auch ein umfangreicher Abhängigkeitsmechanismus zu dem Framework. Ressourcen von Bundles sind standardmäßig nicht nach außen sichtbar. Sie müssen explizit auf Ebene von Java-Packages exportiert werden, damit andere Bundles diese importieren und nutzen können. Durch das statische Importieren werden feste Abhängigkeiten geschaffen. Um die Vorteile der dynamischen Lebenszyklen nutzen zu können, importiert man keine Ressourcen, sondern fragt von der Service Registry Dienste (Java Objekte) ab, die einen bestimmten Vertrag erfüllen bzw. ein bestimmtes Interface implementiert haben. Ein Bundle kann einen Dienst in der Service Registry

45 Siehe o.V.: OSGi Alliance, o.J.

anmelden, andere Bundles können dann diesen Dienst abfragen und die exportierten Funktionen nutzen. Dadurch ist eine lose Kopplung von Bundles möglich.

Dieses Dienste-Konzept ist in Eclipse mit den Extension Points und Extensions realisiert. Ein Plugin kann einen Extension Point bereitstellen, der typischerweise aus einem Java Interface zusammen mit weiteren XML-Angaben besteht. Ein anderes Plugin, das sich zu diesem Extension Point verbinden möchte, muss dieses Interface in seiner Extension implementieren. Das zweite Plugin erweitert somit die Funktionalität des ersten Plugins, ohne dass das erste Plugin genaue Kenntnisse über das zweite hat. Es weiß nichts, was über die geforderten Angaben aus dem Extension Point hinausgeht.

5.2.2 iDocIt!

Wie eingangs beschrieben, soll sich iDocIt! in die Entwicklungsumgebung Eclipse integrieren und deren Komponenten als Eclipse Plugins realisiert werden.

Zur Erweiterung des Werkzeugs um unterstützte Programmier- und Beschreibungssprachen existieren noch weitere Möglichkeiten, neben dem in Kapitel 5.2.1 beschriebenen Konzept der Extension Points und Extensions. Eclipse Plugins können auch durch Plugin Fragments und auch herkömmliche Varianten erweitert werden. Zu den herkömmlichen Varianten zählen das Nachladen von Dateien oder Installieren von immer komplett neuen Programmversionen, um eine neue Programmier- oder Beschreibungssprache hinzuzufügen.

- **Plugin Fragment:** Ein Plugin Fragment ähnelt physisch sehr einem normalen Plugin. Während der Laufzeit integriert es sich aber logisch in ein anderes Plugin (Host-Plugin). Es ist dann so, als wenn nur ein Plugin existieren würde. Dadurch hat das Fragment auch uneingeschränkten Zugriff auf die Ressourcen des Host-Plugins. Ein Fragment dient hauptsächlich zur optionalen Erweiterung eines fest spezifizierten Host-Plugins.

- **Nachladen von Dateien:** Beim Nachladen von Dateien würde man z. B. aus einem definierten Ordner alle oder nur bestimmte Dateien (z. B. jar-Dateien) laden und dann deren bereitgestellte Funktionalität nutzen. Diese Dateien müssen dabei bestimmte Anforderungen (z. B. implementierte Schnittstellen) erfüllen, damit sie auch verwendet werden können.
- **Neue Programmversion:** Bei dieser Variante würde eine bestehende Installation des Programms immer durch eine komplett neue Programmversionen ersetzt werden, um neue Funktionalitäten zu erhalten.

Die Plugin Fragments können zwar dazu verwendet werden Programm-funktionalitäten einem Plugin hinzuzufügen (das Core Plugin könnte der Host für die Parser-Erweiterungen Fragments sein), sie sind aber erstens nicht zu diesem Zweck entwickelt worden. Sie sind eher dazu gedacht abhängig vom Betriebssystem verschiedene Bibliotheken zu laden oder um Sprachversionen zu ermöglichen, also nur kleine Programmveränderungen. Zweitens ist im Zuge der Veröffentlichung von iDocIt! als Open Source-Projekt angedacht, dass Parser-Erweiterungen verteilt entwickelt werden. Dadurch können bei der Verwendung von Fragments Klassenpfadkonflikte auftreten. Unterschiedliche Entwickler können in verschiedenen Parser-Erweiterungen gleich benannte Klassen und gleiche Bibliotheken (vielleicht auch in verschiedenen Versionen) verwenden. Wenn diese Fragments sich dann in das Host-Plugin integrieren, verwenden sie gemeinsam den Classloader des Host-Plugins, der dann den Konflikt feststellt und die Konflikt verursachenden Fragments nicht lädt.[46] Im Gegensatz dazu hat jedes Plugin (Bundle) seinen eigenen Classloader.

Das nachladen von jar-Dateien kann dynamisch realisiert werden, aber warum sollte man das sehr gute Konzept zum dynamischen Laden von Plugins von Eclipse vereinfacht nachprogrammieren, wenn es einem bereits getestet zur Verfügung steht? Weiterhin besteht hier wieder das gleiche Problem wie bei den Fragments, dass

46 Vgl. o.V.: OSGi Service Platform - Core Specification, 2009, „Fragment Bundles“, S. 74ff

Klassenpfadkonflikte auftreten können und zusätzlich Abhängigkeiten (z. B. zu weiteren Bibliotheken) selbst aufgelöst werden müssen. Auch das Konzept von Eclipse, zur Bereitstellung von Plugins auf sogenannten Update-Sites zusammen mit einem komfortablen Assistenten zur Installation- und Deinstallation von Plugins, kann dann nicht genutzt werden.

Bei der immer neu zu installierenden Programmversion ist man jedes Mal gezwungen den vollen Funktionsumfang zu installieren, obwohl man vielleicht nur einen Teil benötigt. Das gleiche gilt, wenn Änderungen an einem Teil der Software vorgenommen werden. Dann kann nicht ein Teil des Programms ausgetauscht werden, sondern es muss immer das gesamte Programm neu installiert werden. Das kann den Umgang mit dem Programm unpraktisch machen. Außerdem ist die Verwaltung von vielen Entwicklern, die an einem großen Plugin arbeiten, schwieriger, als wenn sie (so gut wie möglich) unabhängig kleinere Plugins entwickeln. Auch der Verwaltungsumfang für den Programmcode steigt bei einem großen Programm, wenn immer wieder Teile (neue Parser-Erweiterungen) integriert werden müssen.

Auch wenn es mehrere Möglichkeiten zur Erweiterung von Eclipse Plugins gibt, ist es für iDocIt! sinnvoll das Konzept der Extension Points und Extensions einzusetzen. Dadurch können sämtliche Vorteile, die Plugins und Eclipse bieten, genutzt werden.

Abbildung 5.6 zeigt die Einteilung der Komponenten von iDocIt! unter Verwendung des Konzeptes der Extension Points. Die grafische Oberfläche (`UI`) soll sich als Editor in Eclipse integrieren, die eine feste Abhängigkeit zur `Core` Komponente (Plattform mit Diensten) von iDocIt! erhalten soll. Diese `Core` Komponente ist das Bindeglied zwischen dem `UI` und den Parser-Erweiterungen (`Parser Extension`).

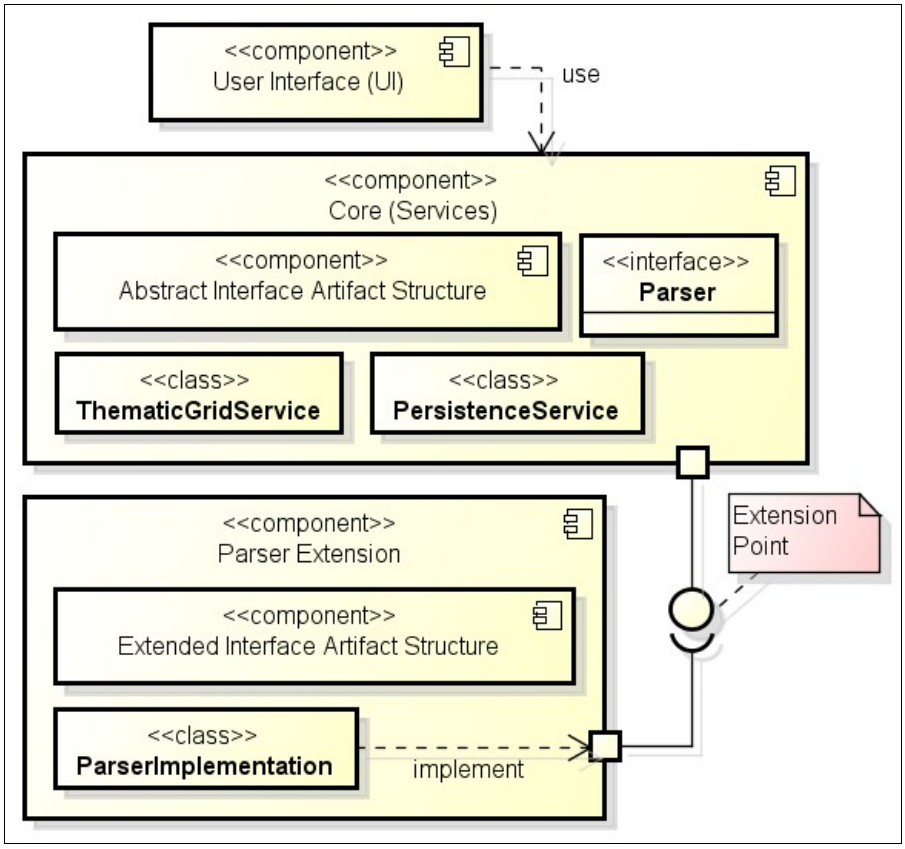


Abbildung 5.6: Komponentendiagramm von iDocIt!

Betrachtet man die einzelnen Komponenten in der Drei-Schichten-Architektur für Softwaresysteme (siehe Abbildung 5.7), so befindet sich das `UI` in der Präsentationsschicht, die `Core` Komponente in der Fachkonzeptschicht und die Parser-Erweiterungen (`Parser Extension`) zur Hälfte in der Fachkonzeptschicht und zur Hälfte in der Datenhaltungsschicht. Zur Vollständigkeit ist in der Datenhaltungsschicht auch ein Artefakt (eine Quelldatei) aufgeführt. Die Parser-Erweiterungen enthalten nicht nur Operationen zum Lesen und Speichern von Daten aus einer Datei, sondern auch die Logik zum Überführen der eingelesenen Schnittstellenstruktur in die allgemeine Schnittstellenstruktur. Deswegen sind sie nicht eindeutig einer Schicht zuordenbar.

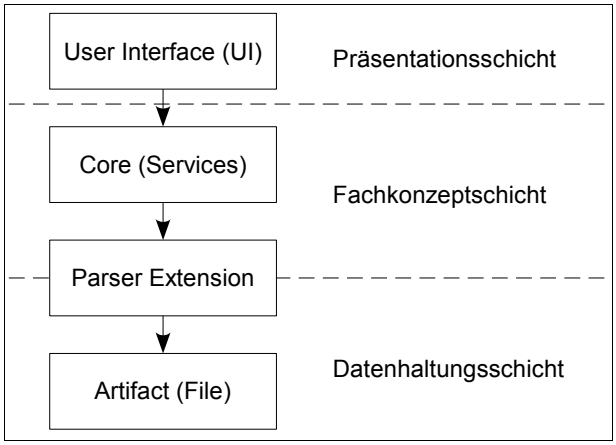


Abbildung 5.7: Einordnung der iDocIt! Komponenten in die Drei-Schichten-Architektur

Die `Core` Komponente stellt für die Kommunikation einige Dienste zur Verfügung. Zum einen in der Klasse `ThematicGridService` (siehe Abbildung 5.8) die Logik für die thematischen Raster, die von Herrn Krause implementiert wurde, als auch Dienste zum Laden und Speichern von Schnittstellendateien in der Klasse `PersistenceService` (siehe Abbildung 5.9).

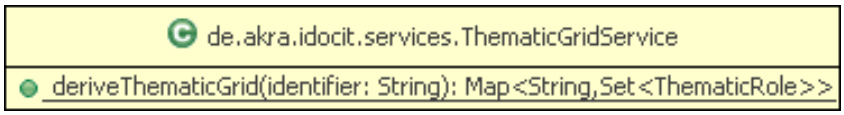


Abbildung 5.8: Klassendiagramm der Klasse `ThematicGridService`

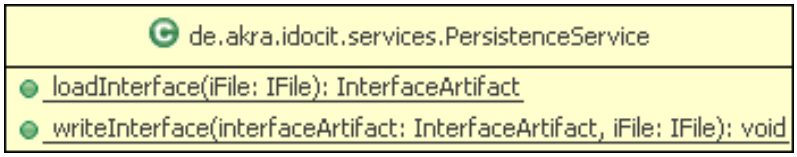


Abbildung 5.9: Klassendiagramm der Klasse `PersistenceService`

Die Methode `deriveThematicGrid(String identifier)` in der Klasse `ThematicGridService` extrahiert aus einem (Operations-) Bezeichner das Verb und gibt die für das Verb definierten thematischen Raster mit ihrer Menge an zugewiesenen thematischen Rollen zurück (`Map<String, Set<ThematicRole>>`). Die Klasse `PersistenceService` bietet die Methode `loadInterface` an. Diese Methode erstellt mit Hilfe der Parser-Erweiterungen aus

unterstützten Eclipse Dateiressourcen (Datentyp: `IFile`) `InterfaceArtifacts`. Mit der Methode `writeInterface` wird ein manipuliertes `InterfaceArtifact`, mit Hilfe einer Parser-Erweiterung, wieder in eine Dateiressource geschrieben.

Auch die abstrakte, allgemeine Schnittstellenstruktur (Abstract `Interface Artifact Structure`), sowie das Interface `Parser` (siehe Abbildung 5.10) sind in der `Core` Komponente enthalten. Dieses Interface ist in dem bereitgestellten Extension Point Vertragsbestandteil.

Es enthält die Methoden

- `parse`: zum Parsen einer Dateiressource und Überführung in ein `InterfaceArtifact`,
- `write`: zum Zurückschreiben eines `InterfaceArtifacts` in eine Dateiressource,
- `isSupported`: zum Prüfen ob der übergeben Dateityp von dem Parser unterstützt wird,
- `getSupportedType`: um den unterstützten Dateityp abzufragen und
- `getDelimiters`: zum Abrufen der von dem Parser verwendeten Trennzeichen für einen Elementpfad (siehe Kapitel 5.3, S. 58).

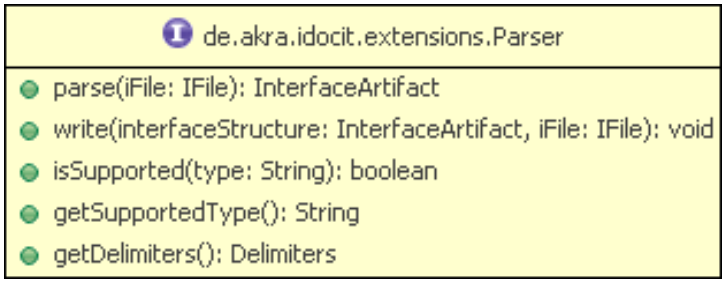


Abbildung 5.10: Klassendiagramm des Interfaces `Parser`

Jede `Parser Extension` Komponente muss das `Parser` Interface implementieren (`ParserImplementation`), um sich als Extension registrieren zu können. Eine Parser-Erweiterung enthält auch die erweiterte, allgemeine Schnittstellenstruktur (`Extended Interface Artifact Structure`), also die von der `Abstract Interface Artifact Structure` erbenden Klassen. Eine `Parser Extension`

Komponente bildet immer einen eigenständigen Parser für eine Programmier- oder Beschreibungssprache.

Das Verhalten der Komponenten wird in Kapitel 5.5 näher beschrieben.

5.3 Parser für Schnittstellen

In Kapitel 5.2.2 wurde bereits gezeigt, dass es für jede unterstützte Programmier- und Beschreibungssprache eine eigene Parser-Erweiterung als Eclipse Plugin geben soll. In diesem Kapitel wird diese Komponente (siehe Abbildung 5.11) mit ihrer allgemeinen Funktionalität detaillierter beschrieben.

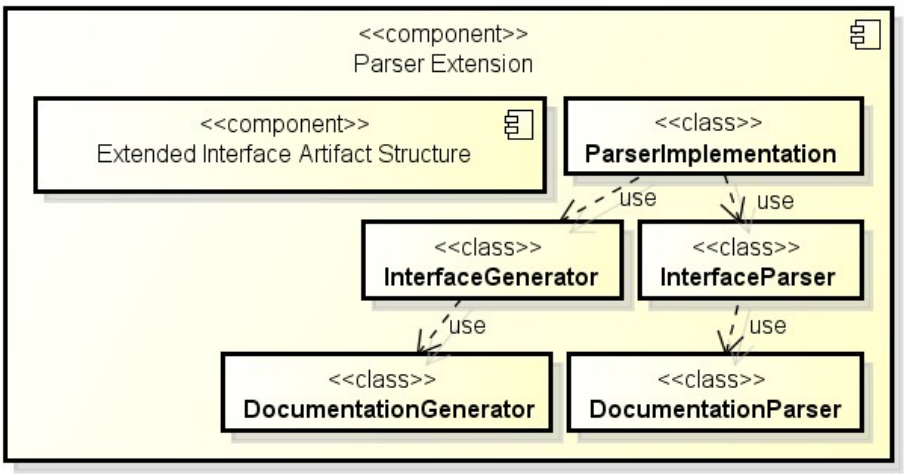


Abbildung 5.11: Komponentendiagramm der `Parser Extension` (siehe auch Abbildung 5.6)

Damit sich die `Parser Extension` Komponente an dem Extension Point der `Core` Komponente registrieren kann, muss sie seinen Vertrag erfüllen. Dazu wird das `Parser` Interface in der Klasse `ParserImplementation` realisiert. In Abbildung 5.12 sind die Parserimplementierungen für Java und WSDL exemplarisch aufgeführt. Einige XML-Angaben müssen von der Extension auch bereitgestellt werden. Es muss das Interface von dem Extension Point angegeben werden und welche Klasse in der Extension dieses Interface implementiert (siehe Tabelle 5.4).

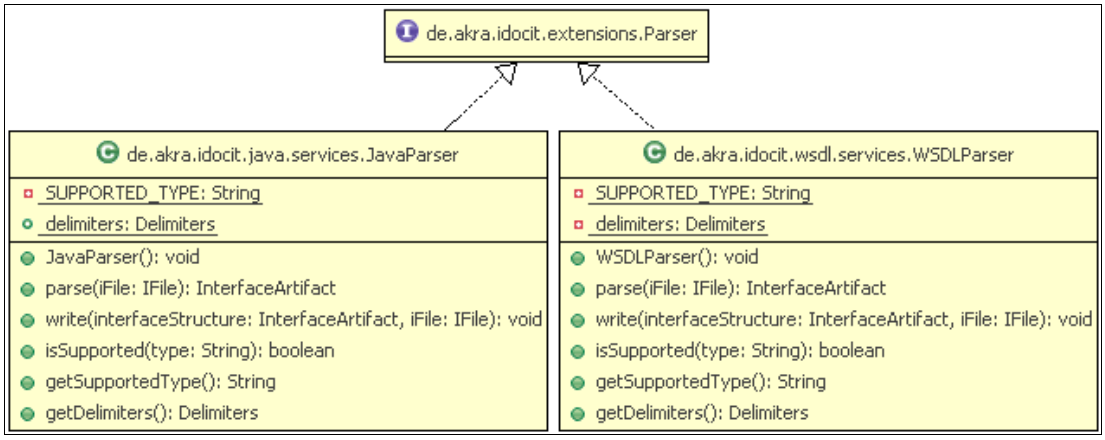


Abbildung 5.12: Klassendiagramm des Interfaces `Parser` mit seinen Kindklassen

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension point="de.akra.idocit.extensions.Parser">
      <client class="de.akra.idocit.wsdl.services.WSDLParser" />
   </extension>
</plugin>
```

Tabelle 5.4: Inhalt der plugin.xml für die WSDL Parser-Erweiterung

Die abstrakte, allgemeine Schnittstellenstruktur, die von der `Core` Komponente zur Verfügung gestellt wird, muss beerbt und bei Bedarf um weitere Attribute ergänzt werden. Die zusätzlichen Attribute sollen alle Informationen enthalten, die für die Rückführung und für das Speichern der Struktur notwendig sind. Diese für eine spezielle Programmier- oder Beschreibungssprache erweiterte Schnittstellenstruktur ist durch die Komponente `Extended Interface Artifact Structure` dargestellt. Im Implementierungskapitel wird die erweiterte Schnittstellenstruktur für WSDL (siehe Kapitel 6.2.3) und Java (siehe Kapitel 6.3.3) vorgestellt.

Die Klasse `ParserImplementation` verwendet u. a. die Klasse `InterfaceParser`, die dafür zuständig ist, einen Quelltext in die allgemeine Struktur zu überführen. Sie soll dazu den Quelltext parsen und nach einer vorher definierten Abbildung einzelne Elemente den Elementen in der allgemeinen Struktur zuordnen und so ein `InterfaceArtifact` aufbauen (siehe Abbildung 5.13). Beim Parsen wird auch jeweils der vorhandene Kommentarblock von einem

Quelltextelement eingelesen und von dem verwendeten `DocumentationParser` in eine Liste von `Documentation`'s zerlegt und dem zugehörigen `SignatureElement` angehängt.

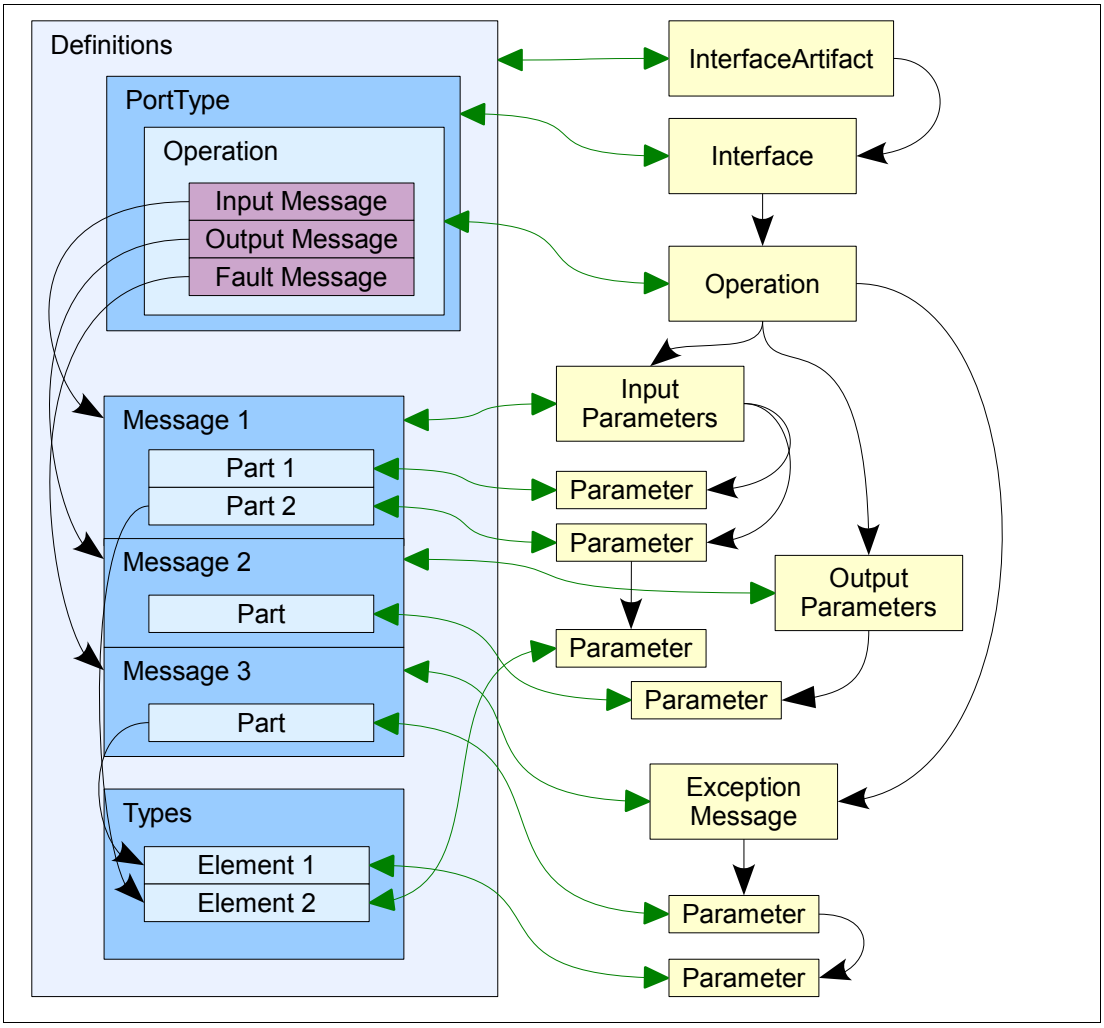


Abbildung 5.13: Beispielhafte Abbildung einer WSDL 1.1 Struktur in die allgemeine Schnittstellenstruktur

Wenn ein `InterfaceArtifact` wieder gespeichert werden soll, dann soll der `InterfaceGenerator` den neuen Quelltext generieren bzw. den alten ändern. Er soll für jedes `SignatureElement` mit Hilfe des `DocumentationGenerator`'s aus einer oder mehreren Listen von `Documentation`'s einen Kommentarblock generieren und diesen zu dem richtigen Element im Quelltext schreiben. Die

Dokumentation eines Artefakts, einer Schnittstelle und einer Operation sollen direkt an das Element geschrieben werden. Die Dokumentation von Eingaben, Ausgaben Fehlern und auch deren Attribute sollen in die Operationsdokumentation integriert werden. Die einzelnen Dokumentationen sollen dabei so gekennzeichnet werden, dass sie in dem Kontext der Operation eindeutig einem Element zugeordnet werden können. Das ist vor allem für das Wiedereinlesen der Dokumentation wichtig, denn dabei müssen die Dokumentationen der einzelnen Elemente wieder den richtigen `SignatureElement`s zugeordnet werden.

Diese Kennzeichnung soll durch eine Pfadangabe von dem Parameter ausgehend zu dem dokumentierten Attribut erfolgen. Der Pfad besteht aus einer Reihe von voll qualifizierten Namen zusammen mit dem jeweiligen ggf. vorhandenen Datentyp. Damit dieser Pfad eindeutig wieder geparst werden kann, sind drei unterschiedliche Trennzeichen nötig, die die einzelnen Informationen voneinander trennen. Ein Trennzeichen um die Elemente in dem Pfad zu trennen, eins um ein Element in den voll qualifizierten Namen und den Datentyp zu teilen und eins um den einfachen Namen (Bezeichner) vom qualifizierenden Teil (Namensraum) zu trennen. Diese Trennzeichen müssen so gewählt werden, dass sie in der jeweils zugrunde liegenden Programmier- oder Beschreibungssprache keine Konflikte verursachen. Dazu können z. B. Zeichen verwendet werden, die in Bezeichnern und Qualifizierungsangaben verboten sind. Da die Trennzeichen variieren können, sollen sie von jeder Parserimplementierung definiert und bereitgestellt werden.

Ein Beispiel für einen Elementpfad in WSDL ist in Tabelle 5.5 aufgeführt. Als Trennzeichen wurden ein Semikolon (;) als Elementtrenner und ein Plus (+) als Trennzeichen zwischen dem qualifizierten Bezeichner des Elements und dem qualifizierten Bezeichner für den Datentyp gewählt. Der qualifizierende Teil (Namensraumangabe) im Bezeichner ist durch das Nummernzeichen (#) von dem einfachen Bezeichner getrennt. Damit der Pfad ein wenig besser lesbar ist, wurde nach jedem Element ein Zeilenumbruch eingefügt.

```
urn:ebay:apis:eBLBaseComponents#GetFeedbackRequest;
urn:ebay:apis:eBLBaseComponents#GetFeedbackRequest+urn:ebay:apis:eB
LBaseComponents#GetFeedbackRequest;
urn:ebay:apis:eBLBaseComponents#GetFeedbackRequest+urn:ebay:apis:eB
LBaseComponents#GetFeedbackRequestType;
urn:ebay:apis:eBLBaseComponents#UserID+urn:ebay:apis:eBLBaseCompone
nts#UserIDType
```

Tabelle 5.5: Pfad von der WSDL Message `GetFeedbackRequest` der Operation `GetFeedback` aus der eBay Trading API bis zum Attribut `UserID` vom Typ `UserIDType`

Die Attribute der Typen von Operationsparametern (Eingaben, Ausgaben und Fehler) sollen soweit berücksichtigt werden, wie sie verfügbar, nach außen sichtbar und nicht rekursiv sind. Wenn sie nicht verfügbar, also deren Deklaration nicht zugreifbar ist, sollen sie ignoriert werden. Das kann z. B. bei der WSDL für Typen passieren, die in einem externen XML-Schema definiert sind. Für Java kann dieses aber z. B. nicht passieren, denn dort müssen alle Typen immer bekannt und verfügbar sein. Wenn Attribute nicht nach außen sichtbar sind, sollen sie auch ignoriert werden. Dieses gilt z. B. für private Attribute in einer Java Klasse, die keine Getter oder Setter Methode haben. Sie werden dann nicht weiter in der allgemeinen Struktur aufgeführt. Da bei der WSDL alle Elemente öffentlich sichtbar sind, hat diese Restriktion dort keinen Effekt.

Geerbte Attribute aus einer Oberklasse sollen in der Struktur wie ein einzelnes Attribut vom Typ der Oberklasse aufgeführt werden. Der Bezeichner für dieses zusätzliche Attribut soll entsprechend der Programmier- bzw. Beschreibungssprache gewählt werden und stets eindeutig unter den Attributen sein (in Java z. B. der Bezeichner `super`, das ein reserviertes Schlüsselwort ist). Wenn der Typ eines Attributs rekursiv definiert ist, soll er nur für den ersten Rekursionsschritt überprüft werden. Die Wiederholungen der Definition sollen nicht in der allgemeinen Struktur berücksichtigt werden, da sie keine neuen Attribut- bzw. Typinformationen liefern.

Ein generelles Problem, das auftreten kann, ist, wenn die Quelldatei (-artefakt) während der Bearbeitung mit iDocIt! von anderen Programmen geändert wird. Hierfür existieren mehrere Lösungen. Um herauszufinden, ob eine Datei geändert

wurde, kann sie überwacht werden, während sie in iDocIt! geöffnet ist. Alternativ kann beim Öffnen der Datei z. B. eine Prüfsumme erzeugt und gespeichert werden, die vor dem Speichern wieder überprüft wird. Mit Hilfe dieser Maßnahmen kann erkannt werden, ob eine geöffnete Datei in der Zwischenzeit geändert wurde. Wenn sie geändert wurde, gibt es wieder mehrere Möglichkeiten, wie darauf reagiert werden kann. Die in iDocIt! gemachten Änderungen können

- verworfen werden und der neue Stand der Datei bleibt erhalten,
- der neue Stand wird mit den Änderungen überschrieben oder
- es wird versucht die Änderungen mit dem neuen Stand zu verschmelzen.

Die ersten beiden Möglichkeiten sind die einfachsten. Für einen Benutzer von iDocIt! wird es aber frustrierend sein, wenn seine Arbeit einfach verworfen wird und er noch einmal neu anfangen muss. Für die zweite Variante ist es noch nicht einmal notwendig zu wissen, ob die Datei in der Zwischenzeit geändert wurde. Beim Öffnen der Datei wird sich der aktuelle Stand gemerkt. Dieser Stand zusammen mit den in iDocIt! gemachten Änderungen wird dann einfach in die Datei zurückgeschrieben und der vorherige Inhalt gelöscht, auch wenn er verändert worden ist. Die dritte Möglichkeit ist die aus Benutzersicht beste Lösung, aber aus programmier-technischer Sicht die schwierigste und aufwendigste. Es müsste ein zeichenweiser oder zeilenweiser Vergleich des neuen Dateistands mit dem alten plus den Änderungen erfolgen. Wenn z. B. in einer Zeile beider Stände Änderungen gemacht wurden, existiert ein Konflikt, den der Benutzer per Hand lösen muss. Zur Konfliktlösung muss der Benutzer dann entweder diesen direkt in der Datei oder über eine eigens dafür programmierten grafischen Oberfläche lösen. Wenn eine Zeile nur in einem Dateistand geändert wurde, kann die Änderung automatisch übernommen werden.

Auch wenn die Verschmelzung von Dateiständen die bessere Lösung ist, ist deren Umsetzung in dieser Bachelorarbeit zu umfangreich. Es soll daher die zweite Variante, das Überschreiben des ggf. geänderten Dateistands mit den Daten aus iDocIt!, realisiert werden.

5.4 User Interface (UI)

Das User Interface (grafische Oberfläche) soll als Eclipse Editor mit Komponenten aus dem Standard Widget Toolkit (SWT) von Eclipse implementiert werden. Das hat den Vorteil, dass es sich in das Konzept und in die Oberfläche von Eclipse nahtlos integriert. Eine Alternative dazu wäre z. B. ein eigenes Fenster basierend auf z. B. Java Swing Komponenten zu realisieren. Dadurch würde aber das Konzept von Eclipse durchbrochen werden und eine nahtlose Integration wäre nicht möglich.

Das SWT ist ein eigenständiges Open Source Toolkit von grafischen Steuerelementen für Java. Es bietet die effiziente und portable Nutzung der vom jeweils genutzten Betriebssystem zur Verfügung gestellten Elemente für grafische Oberflächen.

Der Editor soll in Eclipse über das Kontextmenü einer Datei geöffnet werden können. In dem Editor, der `DocumentationEditor` heißen soll, soll dann die Baumstruktur eines `InterfaceArtifact`'s abgebildet werden und die einzelnen Elemente des Baums dokumentiert werden können. In Abbildung 5.14 ist ein Layoutentwurf für das UI zu sehen. Es wurde nach dem POC-UI (Process Oriented Construction of User Interfaces) Verfahren von der AKRA GmbH entwickelt. „Die POC-UI-Methode ist ein leichtgewichtiges Verfahren zur softwaretechnischen Strukturierung von grafischen Oberflächen.“[47]

Aus den Anforderungen, die erfüllt werden sollen, wurde das zu Grunde liegende Benutzungsmodell entwickelt.

47 Vgl. Krause, Jan Christian / Joneleit, Kai Benjamin: Prozessorientierte Konstruktion von Benutzeroberflächen, 2009, S. 27

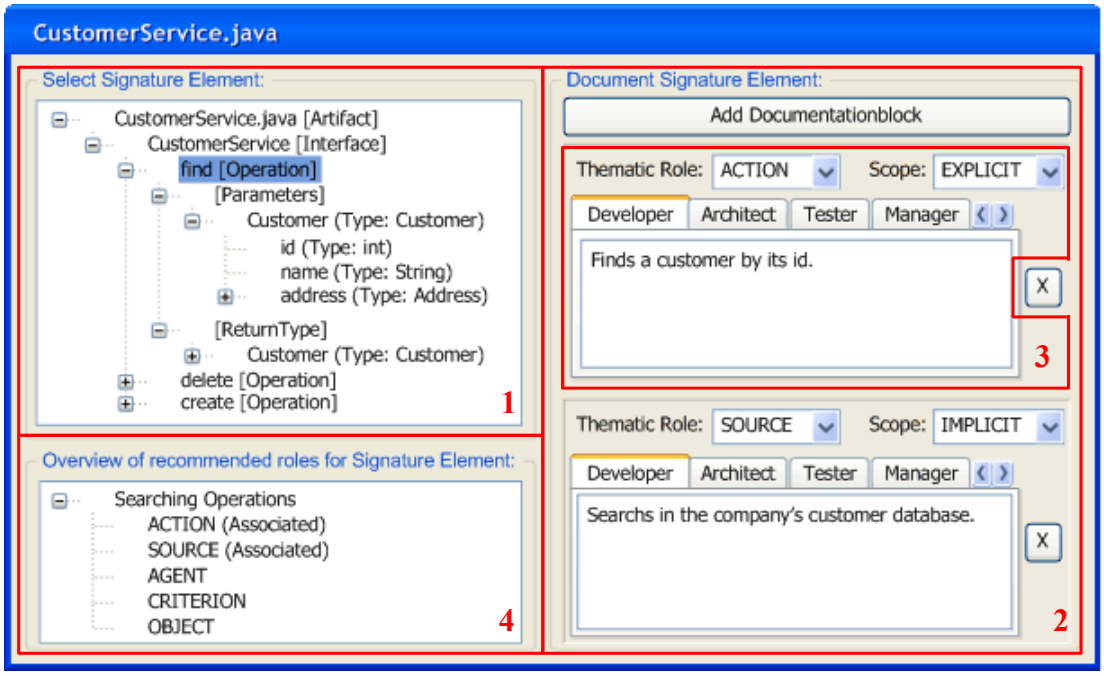


Abbildung 5.14: Layoutentwurf des User Interfaces

Im Folgenden werden die Zuständigkeiten der in Abbildung 5.14 durchnummerierten Komponenten kurz skizziert:

Jedes Element (`SignatureElement`) im `InterfaceArtifact` soll mit beliebig vielen Dokumentationsblöcken dokumentiert werden können. Da das `InterfaceArtifact` eine Baumstruktur bildet, braucht das UI eine Komponente, mit der durch diese Baumstruktur navigiert werden kann. Diese Komponente soll `SelectSignatureElementComposite` (1) heißen.

Wenn auf ein Element in diesem Baum geklickt wird, sollen die bereits vorhandenen Dokumentationsblöcke für dieses `SignatureElement` in einer Liste angezeigt werden. Zu dieser Liste sollen weitere Blöcke hinzugefügt, aber auch vorhandene entfernt werden können. Die Liste und die Funktionen sollen in der Komponente `DocumentationItemListComposite` (2) realisiert werden. In dieser Komponente soll sich an zentraler Stelle eine Schaltfläche befinden, mit der Dokumentationsblöcke der Liste hinzugefügt werden können. Eine Zeile der Liste

besteht aus einem Dokumentationsblock und einer Schaltfläche zum Entfernen der Dokumentation.

Ein Dokumentationsblock soll in einem `DocumentationItemComposite` (3) zusammengefasst werden. Diese Komponente soll dynamische Auswahlmöglichkeiten für eine thematische Rolle und einem Sichtbarkeitsbereich enthalten. Weiterhin soll sie für eine variable Menge an Adressaten jeweils ein Freitextfeld enthalten. Sobald eine Änderung der Dokumentation erfolgt ist, soll die neue Dokumentation über die Speichern-Schaltfläche von Eclipse gespeichert werden können.

Beim Selektieren einer Operation, eines Parameters einer Operation oder eines Attributs eines Parameters im `SelectSignatureElementComposite` soll für das Verb der Operation die hinterlegten thematischen Raster angezeigt werden. Für ein Verb können mehrere thematische Raster mit unterschiedlichen thematischen Rollen definiert worden sein. Daraus ergibt sich die Struktur einer Liste von Rastern mit jeweils einer Liste von Rollen. Diese Struktur soll in der Komponente `DisplayRecommendedRolesComposite` (4) als Baumstruktur angezeigt werden.

Für eine bessere Übersicht sollen in der O-Doku bereits verwendete Rollen eine Kennzeichnung erhalten. Diese Komponente soll keine weitere Funktionalität erhalten, die über das Anzeigen von den thematischen Rastern hinausgeht.

Bereits zugeordnete thematische Rollen für eine Operation sollen in der Dokumentation aller Elemente der Operation gesucht werden, als auch in den Dokumentationen der Elemente auf direktem Weg hoch zum Wurzelelement der Struktur, dem `InterfaceArtifact`. Zu den Elementen einer Operation zählen seine Eingaben, Ausgaben, Ausnahmen und wiederum jeweils den Attributen ihrer Datentypen.

In der beispielhaften Schnittstellenstruktur aus Abbildung 5.15 würden für die Methode `getFeedback` in allen Dokumentationen der zur Methode gehörenden Elemente gesucht werden. Dazu gehören die Methode selbst und alle Ausnahmen

und der Ergebnistyp. Auf dem Weg bis zum Wurzelelement würden noch die Dokumentationen des Elements `GetFeedbackCall` und die Dokumentationen des Wurzelelements `GetFeedbackCall.java` nach bereits zugeordneten Rollen durchsucht werden.

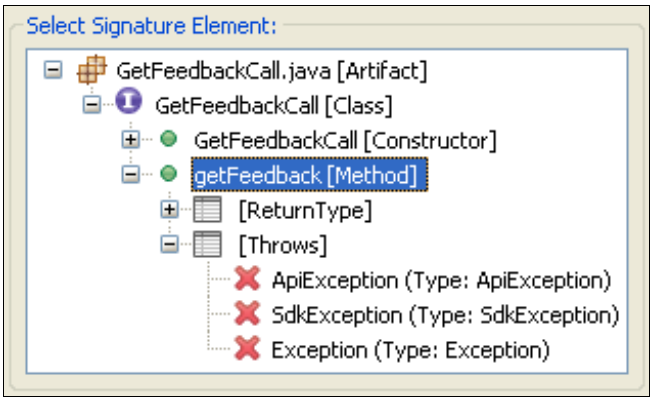


Abbildung 5.15: Beispiel für die Ermittlung von zugeordneten thematischen Rollen

Die Zustände der beschriebenen Komponenten werden in der Komponente `EditArtifactDocumentationComposite` verwaltet, das sich direkt in dem `DocumentationEditor` einbettet (siehe Abbildung 5.16). Die Details zu den Klassen und Zuständen werden in Kapitel 6.4 beschrieben.

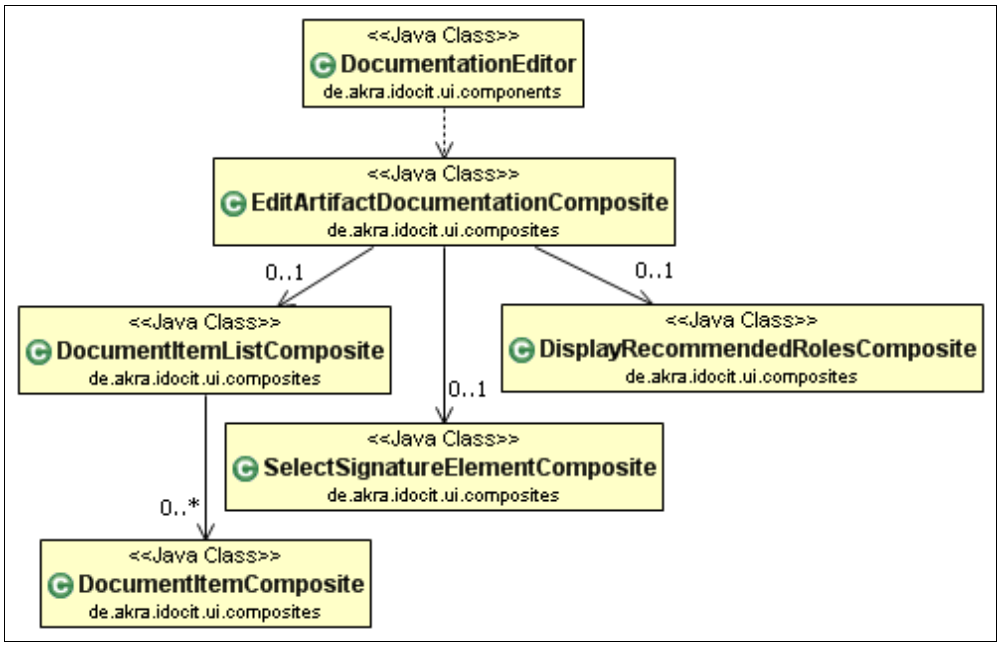


Abbildung 5.16: Klassendiagramm der UI Komponenten

5.5 Kommunikation der Plattform-Services

In diesem Kapitel wird das Zusammenspiel von Eclipse und den Komponenten (Plugins) von iDocIt! anhand von einigen Anwendungsfällen beschrieben.

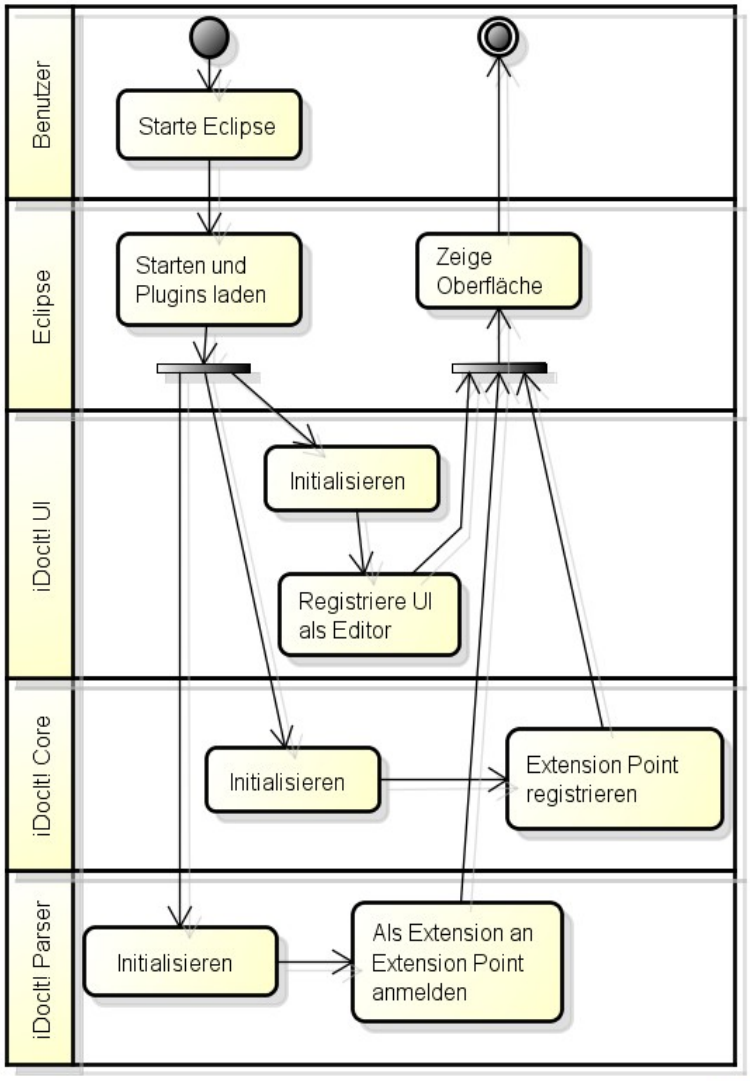


Abbildung 5.17: Aktivitäten beim Start von Eclipse

Damit ein Benutzer iDocIt! nutzen kann, muss er zuerst Eclipse starten (siehe Abbildung 5.17). Beim Start lädt Eclipse u. a. die hinterlegten Plugins. Diese initialisieren sich und registrieren sich bei Eclipse, je nach ihren Einstellungen bzw. ihrem Zweck. Die UI Komponente registriert sich bei Eclipse als Editor, damit sie über das Kontextmenü einer Datei geöffnet werden kann. Die Core Komponente

registriert den Extension Point, damit sich Parser-Erweiterungen an diesen anmelden können. Die Parser-Erweiterungen melden sich dann bei diesem Extension Point als Extension an. Wenn alle Komponenten von Eclipse geladen sind, wird die Oberfläche geöffnet und der Benutzer kann anfangen zu arbeiten.

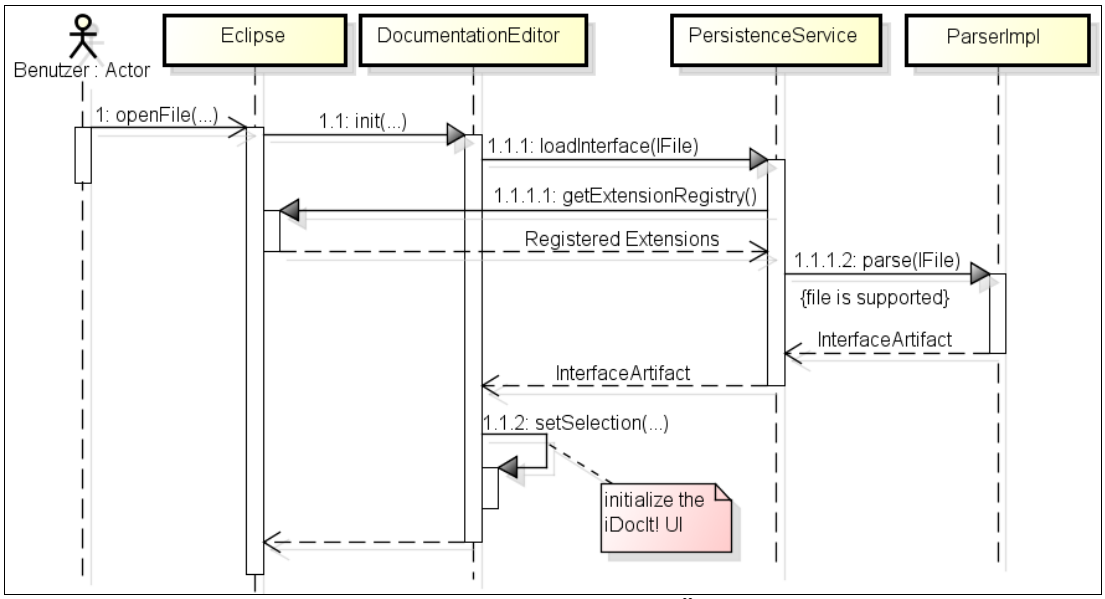


Abbildung 5.18: Ablauf der Methodenaufrufe, beim Öffnen einer Datei in iDocIt!

Wenn der Benutzer eine Datei über dessen Kontextmenü mit iDocIt! öffnet, wird der `DocumentationEditor` (siehe Abbildung 5.16) von iDocIt! initialisiert. Das Sequenzdiagramm in Abbildung 5.18 beschreibt den Ablauf dieser Initialisierung. Dem Editor wird u. a. die Eclipse Dateiressource (Datentyp: `IFile`) übergeben, die er an den `PersistenceService` weitergibt, um die Datei parsen zu lassen. Der `PersistenceService` fragt zuerst von Eclipse die angemeldeten Extensions für das `Parser` Interface ab. Wenn eine Parser Extension für den übergebenen Dateityp vorhanden ist (Prüfung auf Dateiendung), wird das `IFile` weiter an den Parser gegeben. Der Parser parst die Datei und überführt die Struktur in die allgemeine Schnittstellenstruktur. Das erstellte `InterfaceArtifact` wird dann an den `PersistenceService` zurückgegeben. Der `PersistenceService` gibt das Objekt weiter an den `DocumentationEditor`, in dem dann das

`EditArtifactDocumentationComposite` mit seinen Komponenten initialisiert wird. Danach wird das Editorfenster in Eclipse angezeigt.

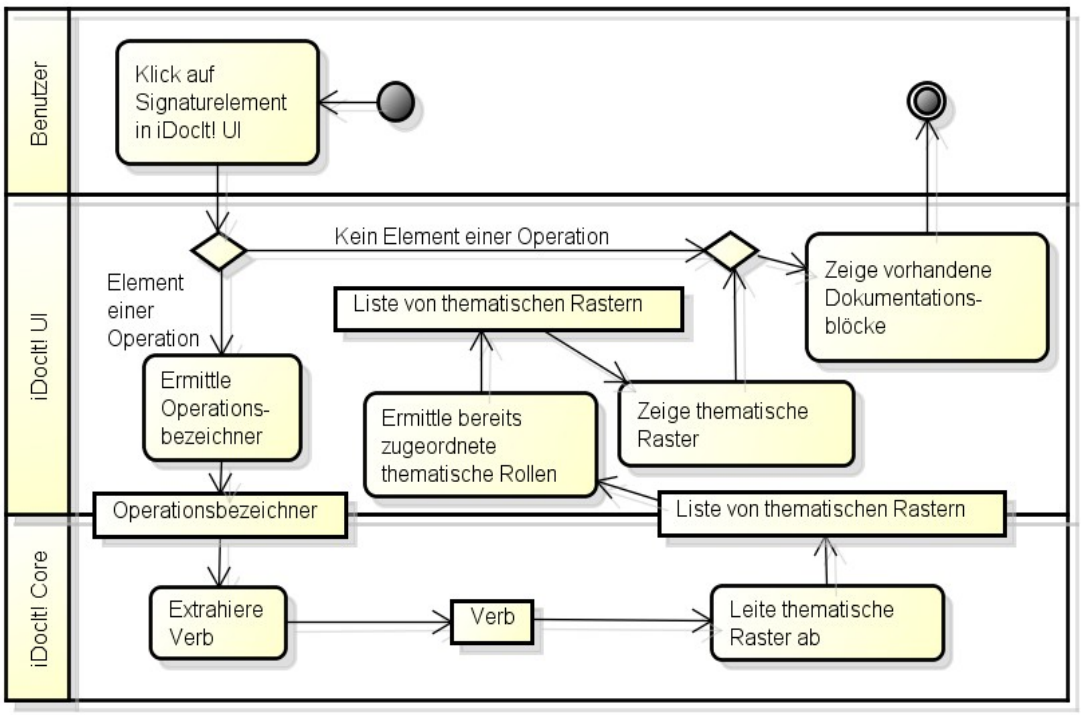


Abbildung 5.19: Aktivitätsdiagramm zum Funktionsablauf, wenn ein Benutzer ein Signaturelement im UI selektiert

Nun kann der Benutzer anfangen die einzelnen Signaturelemente der Schnittstelle zu dokumentieren. Beim Klick auf eines der Signaturelemente in dem Elementbaum (`SelectSignatureElementComposite`) wird das Element erst einmal überprüft, ob es ein Element einer Operation ist oder nicht (siehe Abbildung 5.19). Zu den Elementen einer Operation zählen die Operation selbst, Eingaben, Ausgaben, Ausnahmen und die Attribute der Typen der Eingaben, Ausgaben und Ausnahmen. Wenn es eines dieser Elemente ist, wird der Operationsbezeichner an die Core Komponente (`ThematicGridService`) weitergegeben. Dort werden die thematischen Raster für das im Bezeichner enthaltene Verb abgeleitet. Für jede thematische Rolle in den Rastern wird geprüft, ob diese bereits innerhalb der Operationselemente oder in einem seiner Elternelemente zugeordnet worden sind.

Wenn es wenigstens einmal zugeordnet worden ist, erhält es eine Kennzeichnung in der Oberfläche. Dann werden die thematischen Raster in der Oberfläche angezeigt (im `DisplayRecommendedRolesComposite`). Wenn das selektierte Signaturelement bereits dokumentiert wurde, werden die Dokumentationsblöcke in der Komponente `DocumentationItemListComposite` aufgelistet.

Wenn die Änderungen im iDocIt! UI gespeichert werden sollen, kann der Benutzer auf die Speichern-Schaltfläche von Eclipse klicken. Das Sequenzdiagramm in Abbildung 5.20 beschreibt den internen Ablauf von iDocIt! zum Speichern der Änderungen in die Quelldatei.

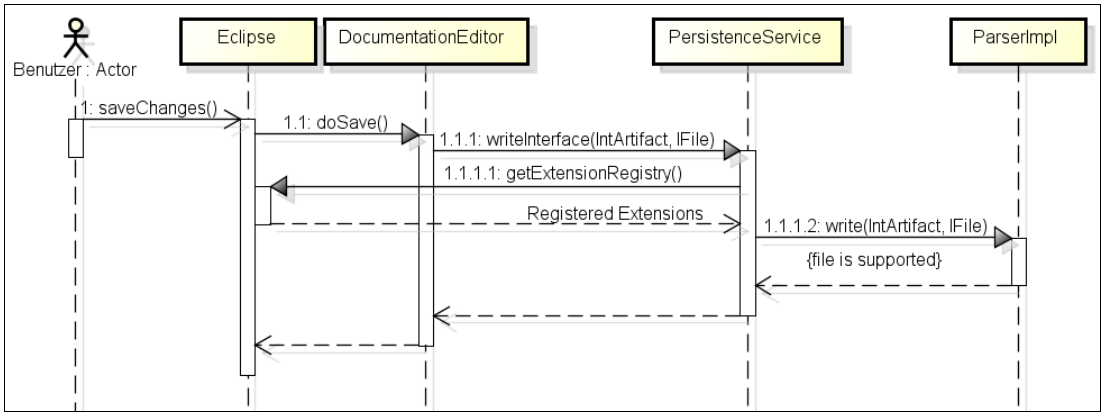


Abbildung 5.20: Ablauf der Methodenaufrufe, beim Speichern von Änderungen

Eclipse ruft zum Speichern der Änderungen im `DocumentationEditor` die Methode `doSave` auf. Dort wird die Methode `writeInterfaceArtifact` des `PersistenceService`'s aufgerufen und die im `DocumentationEditor` gespeicherte Dateiressource (Datentyp: `IFile`) und das geänderte `InterfaceArtifact` mitgegeben. Der `PersistenceService` ruft dann von Eclipse wieder alle registrierten Parser Extensions ab und wenn ein passender Parser dabei ist, wird dem das `InterfaceArtifact` und `IFile` übergeben, damit dieser es in die Dateiressource speichern kann.

6 Implementierung

Dieses Kapitel beschreibt die Realisierung der einzelnen Komponenten von iDocIt!. Zuerst wird die Plattform (Core) mit der abstrakten allgemeinen Schnittstellenstruktur beschrieben. Danach wird die Implementierung der Parser-Erweiterungen für WSDL und Java erläutert und abschließend die der grafischen Oberfläche.

6.1 Plattform (Core)

Die Plattform enthält die abstrakte, allgemeine Schnittstellenstruktur und bietet Dienste zum Laden und Speichern eines Schnittstellenartefakts, sowie einen Dienst zum Ableiten von thematischen Rastern aus einem Operationsbezeichner an. In diesem Kapitel sollen einige Details zu dieser Struktur und der Plattform erläutert werden.

6.1.1 Abstrakte Schnittstellenstruktur

Wie bereits in Kapitel 5.1.2 erwähnt, ist ein `SignatureElement` (siehe Abbildung 5.5, S. 47) nicht eindeutig durch seine Attribute identifizierbar, weswegen ein technischer Schlüssel als das Attribut `id` eingeführt wurde. Benötigt wird dieser eindeutige Schlüssel für die Verwaltung von Informationen zu einzelnen `SignatureElement`s außerhalb der Struktur des `InterfaceArtifact`s. Beispielsweise werden in der grafischen Oberfläche von den Dokumentationsblöcken eines `SignatureElement`s die zuletzt aktiven Adressatenreiter gespeichert. Dadurch können diese bei erneuter Auswahl des `SignatureElement`s wieder vorausgewählt werden.

Ohne den technischen Schlüssel könnten die gespeicherten Daten für ein `SignatureElement` nicht eindeutig wiedergefunden werden. Ein Datentyp kann beispielsweise für mehrere verschiedene Operationsparameter verwendet werden. Aus den Attributen dieses Typs resultieren immer die gleichen

`SignatureElements` bzw. `Parameter`. Wenn man diese Objekte isoliert miteinander vergleicht, sind sie identisch. Unter Einbeziehung ihrer Kontexte, also die Umgebung der Elemente bis zum Wurzelelement, sind sie wieder eindeutig identifizierbar.

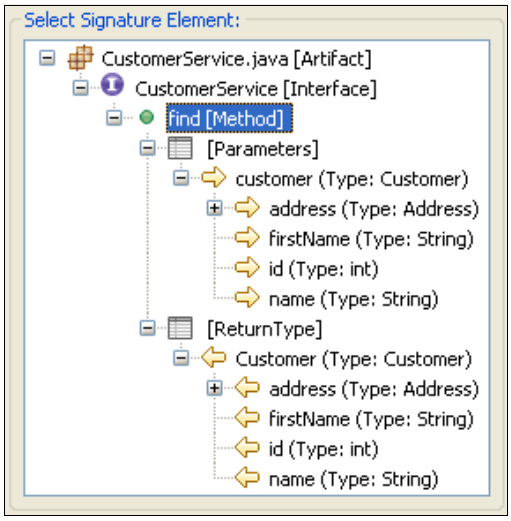


Abbildung 6.1: Beispiel der Wiederverwendung des Datentyps `Customer`

In Abbildung 6.1 ist dieses beispielhaft an dem Datentyp `Customer` und seinen Attributen gezeigt. Der Datentyp wird einmal als Eingabeparameter und einmal als Ergebnistyp verwendet. Die einzelnen Attribute des Typs bilden jeweils ein `SignatureElement`. Z. B. ist das Attribut „`name (Type: String)`“ im Kontext des Eingabeparameters und einmal im Kontext des Ergebnistyps vorhanden. Es hat beide Male die gleichen Attribute und ist somit isoliert betrachtet identisch. Wenn der gesamte Kontext dieses Elements betrachtet wird, also der Pfad bis hoch zum Wurzelelement „`CustomerService.java [Artifact]`“, dann ist es wieder eindeutig. Es würden die Pfade

- „`CustomerService.java/CustomerService/find/`**`c`**`ustomer/name`“
- „`CustomerService.java/CustomerService/find/`**`C`**`ustomer/name`“

entstehen. Man beachte den Unterschied, dass einmal der Parameterbezeichner `customer` und einmal der Ergebnistyp `Customer` in dem Pfad, der den Kontext der Elemente beschreibt, vorkommt.

Eine aktive Einbeziehung der Elementumgebung während der Laufzeit ist nicht möglich, da die Struktur doppelt verkettet ist. Z. B. bei einem Test auf Gleichheit eines Elements mit einem anderen, würde ein Element seine Umgebung, also wegen der Doppelverkettung seine Vorgänger und seine Nachfolger, in die Gleichheitsprüfung mit einbeziehen. Die Vorgänger und Nachfolger verwenden den gleichen Algorithmus. Somit würden sie auch das Ausgangselement mit in ihre Gleichheitsprüfung einbeziehen, wodurch alles wieder von vorne beginnen würde. Der Gleichheitstest würde daher in einer Endlosschleife enden.

Eine Alternative wäre die passive Bestimmung der Elementumgebung. Dabei würde jedes `SignatureElement` ein Attribut erhalten, das den Kontext beschreibt, beispielsweise den Pfad vom `InterfaceArtifact` (die Wurzel) bis zum Element. Dann wäre jedes `SignatureElement` mit diesem Attribut eindeutig identifizierbar. Die Nutzung einer ID bezweckt das gleiche. Dabei hat ein Attribut vom Java Datentyp `int` die großen Vorteile, dass es sehr viel performanter und speicherplatzsparender ist (benötigt vier Byte) als eine Zeichenkette (Java `String`), in der jedes Zeichen zwei Byte groß ist plus den Overhead des Objekts.

Das `SignatureElement` enthält alle beschreibenden Attribute für ein Element aus der allgemeinen Struktur. Die Klassen `InterfaceArtifact`, `Interface`, `Operation` und `Parameters` enthalten jeweils nur noch Attribute, die die Struktur des Elements beschreiben. Lediglich die Klasse `Parameter` hat neben der Liste für enthaltene `Parameter` Instanzen (`complexType`) noch die Attribute für einen Datentypen und einen Pfad vom Parameter einer Operation bis zum entsprechenden Attribut. Dieser Pfad wird benötigt, da die Dokumentationen von Parametern und deren Attribute in die O-Doku integriert wird. Beim Einlesen der O-Doku werden die

Dokumentationsblöcke anhand dieses Pfades wieder eindeutig den Parametern bzw. Attributen der Parameter zugeordnet.

Alle von `SignatureElement` erbenden Klassen sind wiederum als abstrakt deklariert. Dadurch kann ein Entwickler einer Parser-Erweiterung bei der Implementierung besser geleitet werden. Er ist gezwungen die Klassen der Struktur zu beerben und durch abstrakte Methoden wird ihm ein Teil der Stellen aufgezeigt, die für seinen Anwendungsfall implementiert werden müssen.

Dieses Prinzip wird beispielsweise bei der Erstellung einer tiefen Kopie eines `SignatureElement`s angewandt. Eine solche Kopie eines `InterfaceArtifact`s wird beispielsweise beim Öffnen einer Datei angelegt. Durch Vergleichen der Tiefenkopie des Initialzustands mit dem aktuellen Zustand kann ermittelt werden, ob der Benutzer Änderungen am `InterfaceArtifact` vorgenommen hat. Ist dies der Fall, so wird die Speichern-Schaltfläche von Eclipse aktiviert.

Alle direkt von `SignatureElement` erbenden Klassen überschreiben die öffentliche Methode `copy`. In deren Implementierung wird als erstes immer die überschriebene Methode in `SignatureElement` auf (`super.copy(parent)`). Der Parameter `parent` ist dabei das (neue) Elternobjekt für die Kopie des `SignatureElement`'s. Damit dieser Ablauf nicht manipuliert werden kann, muss die Methode in allen von `SignatureElement` erbenden Klassen als `final` (nicht weiter überschreibbar) deklariert werden. Das Codebeispiel in Tabelle 6.1 zeigt dieses an der Klasse `Parameters`.

In der `copy` Methode des `SignatureElement`'s wird dann zuerst die abstrakte Methode `createSignatureElement` aufgerufen (siehe Tabelle 6.2). Diese Methode ist nicht in der Klasse `Parameters`, sondern in der von `Parameters` erbenden Klasse implementiert, z. B. in der Klasse `WSDLMessage` aus der WSDL Parser-Erweiterung (siehe Tabelle 6.3). Diese Klasse repräsentiert ein `<message>`-Element in der WSDL. In dieser Methode wird ein neues Objekt der Subklasse

(`WSDLMessage`) erzeugt und zurückgegeben. Danach werden in `copyAttributesTo` alle Attribute des `SignatureElement`'s und in `doCopyTo` alle Attribute der von `Parameters` erbenden Klasse in das `newSigElem` kopiert. Die abstrakte Methode `doCopyTo` muss auch wieder in der von `Parameters` erbenden Klasse (`WSDLMessage`) entsprechend implementiert werden. Wenn das erledigt ist, werden auch die Attribute und somit alle Nachfolger der Klasse `Parameters` kopiert.

```
public abstract class Parameters extends SignatureElement {

  @Override
  public final SignatureElement copy(SignatureElement parent) {
    SignatureElement newElement = super.copy(parent);
    Parameters newParameters = (Parameters) newElement;

    for (Parameter param : this.parameters) {
      paramList.addParameter(
          (Parameter) param.copy(newParameters));
    }
    return newParameters;
  }
  ...
}
```

Tabelle 6.1: Auszug der Klasse `Parameters`

```
public abstract class SignatureElement {

  public SignatureElement copy(SignatureElement parent) {
    SignatureElement newSigElem = createSignatureElement(parent);
    copyAttributesTo(newSigElem);
    doCopyTo(newSigElem);
    return newSigElem;
  }
  ...
}
```

Tabelle 6.2: Auszug der Klasse `SignatureElement`

```
public class WSDLMessage extends Parameters {

  @Override
  protected SignatureElement
  createSignatureElement(SignatureElement parent) {
    return new WSDLMessage(parent, super.getCategory());
  }

  @Override
  protected void doCopyTo(SignatureElement signatureElement) {
    WSDLMessage wsdlMessage = (WSDLMessage) signatureElement;
    // copy attributes to wsdlMessage ...
  }
  ...
}
```

Tabelle 6.3: Auszug der Klasse `WSDLMessage`

6.1.2 Parser Extension Point

Plugins können in Eclipse während der Laufzeit installiert, gestartet, gestoppt und deinstalliert werden. Die Parser-Erweiterungen können sich demnach während der Laufzeit an dem Extension Point der Plattform an- und abmelden. Daher muss diese darauf reagieren können.

Eine Möglichkeit wäre die angemeldeten Parser Extensions einmal von Eclipse abzufragen und zwischenzuspeichern. Dann muss aber auch ein Listener bei Eclipse registriert werden. Der Listener reagiert dann auf neue und deinstallierte Extensions und aktualisiert den Zwischenspeicher. Eine andere Möglichkeit wäre die Extensions jedes Mal neu von Eclipse abzufragen, wenn sie benötigt werden.

Für diese Bachelorarbeit wurde die zweite Variante realisiert. Eine Parser Extension wird nur beim Laden und Speichern einer Datei benötigt. Daher fallen die wenigen Aufrufe, die dadurch mehr entstehen, als wenn sie zwischengespeichert werden würden, nicht ins Gewicht. Dadurch wird aber der zusätzliche Verwaltungsaufwand des Zwischenspeichers gespart.

6.2 Parser für WSDL

Die Parser-Erweiterung für WSDL ist unter der geforderten Prämisse entwickelt worden, dass die Quelldatei nur um die Dokumentation verändert wird. D. h. dass der vorhandene Inhalt bestehen bleiben soll, sodass der Programmcode nicht verändert wird. Unterschiedliche Formatierungen (z. B. Leerzeichen und Zeilenumbrüche) verändern bei der WSDL nicht den Programmcode, da sie auf XML basiert. Eine Veränderung des Formats ist demnach nicht entscheidend, aber sie sollte trotzdem bestmöglich vermieden werden. In z. B. einer Versionskontrolle sollen bei einem Dateivergleich nur die wichtigen Änderungen berücksichtigt werden. Unnötige Formatänderungen sollen diese nicht überlagern und auch nicht zu unnötig vielen Dateiänderungen führen.

Von der Parser-Erweiterung werden nur bestimmte Teile der WSDL Elemente aus der allgemeinen Schnittstellenstruktur zugeordnet. Die restlichen Teile müssen aber trotzdem eingelesen und behandelt werden, damit die Datei vollständig geschrieben werden kann. Nicht zugeordnet werden z. B. das `<service>`- und `<binding>`-Element in einer WSDL Datei. Um den Entwicklungs- und Implementierungsaufwand für einen vollständigen Parser für WSDL mit zurückschreiben der Daten zu vermeiden, wird auf einen existierenden Parser zurückgegriffen.

Zunächst werden einige existierende Parser evaluiert, ob deren Funktionalität für den Zweck WSDL zu parsen und zu speichern ausreicht. Daran folgend wird die XML Struktur der Dokumentationen von iDocIt! entwickelt, wie sie in der WSDL (im `<documentation>`-Element) gespeichert werden soll. Zum Schluss werden einige Details zur Implementierung beschrieben.

6.2.1 Existierende XML / WSDL Parser

Da die WSDL auf XML basiert, können XML Parser verwendet werden, um dessen Elemente zu parsen. Es existieren für Java viele solcher Parser, die eine XML Struktur parsen können, wie z. B. der Xerces2 Java Parser[48] oder der `DocumentBuilder` und `SAXParser` aus dem `javax.xml.parsers` Paket der Java Platform (Standard Edition 6)[49]. Es existieren auch Parser, die eine WSDL parsen und eine Java Objektstruktur als Ergebnis ausgeben. Beispiele dafür sind WSDL4J[50] und das Membrane SOA Model[51].

Der `DocumentBuilder` und `SAXParser` sind reine XML Parser und können die geparste Struktur nicht wieder zurückschreiben. Der Xerces2 Java Parser bietet dagegen mit dem `LSSerializer` (aus dem Paket `org.w3c.dom.ls`) die Möglichkeit ein DOM (Document Object Model) Dokument auch zu schreiben. Aber beim Parsen müssen per Hand die eingelesenen Elemente in eine Objektstruktur überführt werden. Änderungen müssen dann wieder in das DOM Dokument eingepflegt werden, damit sie gespeichert werden können.

WSDL4J und das Membrane SOA Model bieten direkte Unterstützung der WSDL an. Sie parsen eine WSDL Datei und erstellen daraus eine Java Objektstruktur. Diese Objektstruktur kann dann manipuliert und wieder gespeichert werden. Das Membrane SOA Model ist zwar kostenlos nutzbar, aber deren API ist nicht dokumentiert und deren Quellen auch nicht öffentlich zugänglich. WSDL4J ist quelloffen und auch dokumentiert, weswegen die Entscheidung letztendlich auf WSDL4J gefallen ist.

48 Vgl. o.V.: Xerces2 Java Parser, o.J.
49 Vgl. o.V.: Java Platform, Standard Edition 6, o.J., „Package javax.xml.parsers“
50 Vgl. o.V.: WSDL4J, o.J.
51 Vgl. o.V.: Membrane SOA Model - Java API for WSDL, o.J.

6.2.2 Struktur der Dokumentationen in WSDL

Dokumentationen von iDocIt! sollen in die optionalen `<documentation>`-Elemente des `<definitions>`-Elements und der einzelnen Elemente vom `<portType>` eingefügt werden können. In einem `<documentation>`-Element darf Freitext aber auch eine Elementstruktur enthalten sein. In einem Dokumentationsblock müssen dabei eine thematische Rolle, ein Sichtbarkeitsbereich, ein Pfad zu dem dokumentierten Element und eine Menge von Adressaten mit jeweils einem Text untergebracht werden.

Wie eingangs erwähnt, basiert die WSDL auf XML. Deswegen soll keine neue Auszeichnungssprache für die iDocIt!-Dokumentationen entwickelt werden, sondern die Möglichkeiten von XML genutzt werden. Die Dokumentationen integrieren sich so nahtlos in die Struktur der WSDL.

Ein Dokumentationsblock wird in einem `<docpart>`-Element realisiert. Das `<docpart>`-Element erhält die Attribute `role` für die thematische Rolle, `scope` für den Sichtbarkeitsbereich und `signatureElement` für den Pfad zum dokumentierten Element. Die Werte der Attribute `role` und `signatureElement` sind die des primitiven Datentyps `string` aus der XML Schema Definition.[52] Das Attribut `scope` darf nur die Werte „EXPLICIT“ oder „IMPLICIT“ enthalten. Zur Darstellung der adressatenspezifischen Beschreibungen enthält das `<docpart>`-Element eine Menge von `<addressee>`-Elementen mit dem `string`-Attribut `group`, das den Adressatennamen enthält. Der in diesem Element enthaltene Text ist die Beschreibung für diesen Adressaten (siehe Tabelle 6.4).

52 Vgl. Biron, Paul V / Malhotra, Ashok: XML Schema Part 2, 2001, „3.2 Primitive datatypes“

```
<wsdl:documentation>
  <docpart role="xsd:string"? scope="EXPLICIT | IMPLICIT"
   signatureElement="xsd:string"?> *
    <addressee group="xsd:string">....</addressee> *
  </docpart>
</wsdl:documentation>
```

Tabelle 6.4: `<documentation>`-Elementstruktur für Dokumentation von thematischen Rollen und verschiedenen Adressaten[53]

Abbildung 6.2 zeigt im Vergleich dazu die Umsetzung im iDocIt! User Interface. Dort kann eine thematische Rolle und ein Sichtbarkeitsbereich ausgewählt, sowie eine Menge von adressatenspezifischen Beschreibungen erfasst werden.

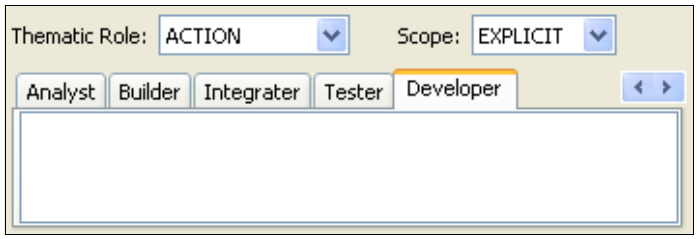


Abbildung 6.2: Dokumentationsblock im iDocIt! UI

53 Legende: ? : Optional (0 bis 1 Mal vorhanden)
* : Mehrfach vorhanden (0 bis beliebig oft)
| : Alternative von Werten

6.2.3 Implementierungsdetails des WSDL-Parsers

WSDL4J in der aktuellen Version 1.6.2 unterstützt nur die WSDL 1.1 Spezifikation. Das ist aber für den Rahmen dieser Bachelorarbeit ausreichend, da nur die Unterstützung von WSDL 1.1 gefordert ist.

Bei der Arbeit mit WSDL4J haben sich aber drei Mängel herausgestellt.

1. Mangel

Zum einen ein nicht so gravierender Mangel, dass die Formatierungen (z. B. Leerzeichen und Zeilenumbrüche) der Quelldatei beim Speichern geändert wird. Das Speichern der WSDL4J Objektstruktur ist in dem Toolkit sehr einfach implementiert. Die fest definierten bekannten Elemente (z. B. `<definitions>`, `<types>`, `<portType>` und `<operation>`) werden nach einem festen Schema formatiert. Beispielsweise sind Zeilenumbrüche und Einrückungen (mit zwei Leerzeichen) hart codiert. Nicht bekannte Elemente, wie z. B. das `<docpart>`- und `<addressee>`-Element werden ohne Zeilenumbrüche und extra Einrückung im `<documentation>`-Element geschrieben.

Daher wurde der Quellcode von WSDL4J angepasst. Die `<docpart>`-Elemente werden nun eine Ebene tiefer eingerückt als das `<documentation>`-Element und auch die Art (z. B. Leer- oder Tabulatorzeichen) und jeweilige Anzahl der Einrückungen kann nun wenigstens über eine globale Konstante angepasst werden. Weitere Anpassungen, wie z. B. Zeilenumbrüche für `<addressee>`-Elemente stehen noch aus.

2. Mangel

WSDL4J ignoriert alle XML-Kommentare beim Parsen der WSDL. Dadurch werden diese beim Zurückschreiben der WSDL gelöscht. Für diese Bachelorarbeit ist der Mangel noch tragbar, aber das Verhalten wäre für den produktiven Einsatz in der Praxis nicht haltbar.

3. Mangel

WSDL4J berücksichtigt beim Einlesen der WSDL nur fest definierte Attribute eines Elements, andere werden ignoriert. In Tabelle 6.5 sind die für diese Bachelorarbeit wichtigen WSDL Elemente mit ihren unterstützten Attributen aufgeführt. Dadurch können wichtige Informationen beim Einlesen verloren gehen, die beim Schreiben nicht mehr berücksichtigt werden können. In dem aktuellen Prototyp von iDocIt! ist dieses Problem noch nicht behoben worden, was aber in einer nachfolgenden Version unbedingt gemacht werden sollte.

Element	**Attribute**
`definitions`	`name, targetNamespace, xmlns`
`portType`	`name`
`operation`	`name, parameterOrder`
`input`	`name, message`
`output`	`name, message`
`fault`	`name, message`
`message`	`name`
`part`	`name, element, type`

Tabelle 6.5: Auszug von WSDL Elementen mit den von WSDL4J unterstützten Attributen

In Kapitel 5.1.2 wurden die für iDocIt! wichtigen Teile der WSDL 1.1 Spezifikation bereits vorgestellt. In Tabelle 6.6 ist die Spezifikation dieser Teile noch einmal zusammenhängend aufgeführt, damit die Zuordnung zu den Klassen der allgemeinen Schnittstellenstruktur gezeigt werden kann. Die Parser-Erweiterung soll vom `<definitions>`-Element das `<portType>`-, `<message>`- und `<types>`-Element analysieren. Dokumentationen werden aber nur in das `<definitions>`-Element und die einzelnen Elemente vom `<portType>` eingefügt.

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
  <wsdl:documentation .... /> ?

  <wsdl:types> ?
    <wsdl:documentation .... /> ?
    <xsd:schema .... /> *
  </wsdl:types>

  <wsdl:message name="nmtoken"> *
    <wsdl:documentation .... /> ?
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </wsdl:message>

  <wsdl:portType name="nmtoken"> *
    <wsdl:documentation .... /> ?
    <wsdl:operation name="nmtoken"> *
      <wsdl:documentation .... /> ?
      <wsdl:input name="nmtoken"? message="qname"> ?
        <wsdl:documentation .... /> ?
      </wsdl:input>
      <wsdl:output name="nmtoken"? message="qname"> ?
        <wsdl:documentation .... /> ?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname"> *
        <wsdl:documentation .... /> ?
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Tabelle 6.6: Auszug aus WSDL 1.1 Spezifikation[54,55]

Das Analysieren und Überführen der Elemente in eine Objektstruktur übernimmt WSDL4J. Aus der resultierenden Objektstruktur wird dann die `InterfaceArtifact` Struktur aufgebaut (siehe Abbildung 6.3). Damit die Dokumentationen später wieder an die richtigen Stellen in der WSDL4J-Objektstruktur geschrieben werden, erhalten alle Klassen, außer die Klasse `WSDLParameter`, eine Referenz zu dem korrespondierenden Objekt.

54 Legende: ?: Optional (0 bis 1 Mal vorhanden); *: Mehrfach vorhanden (0 bis beliebig oft)
55 Vgl. Anhang B

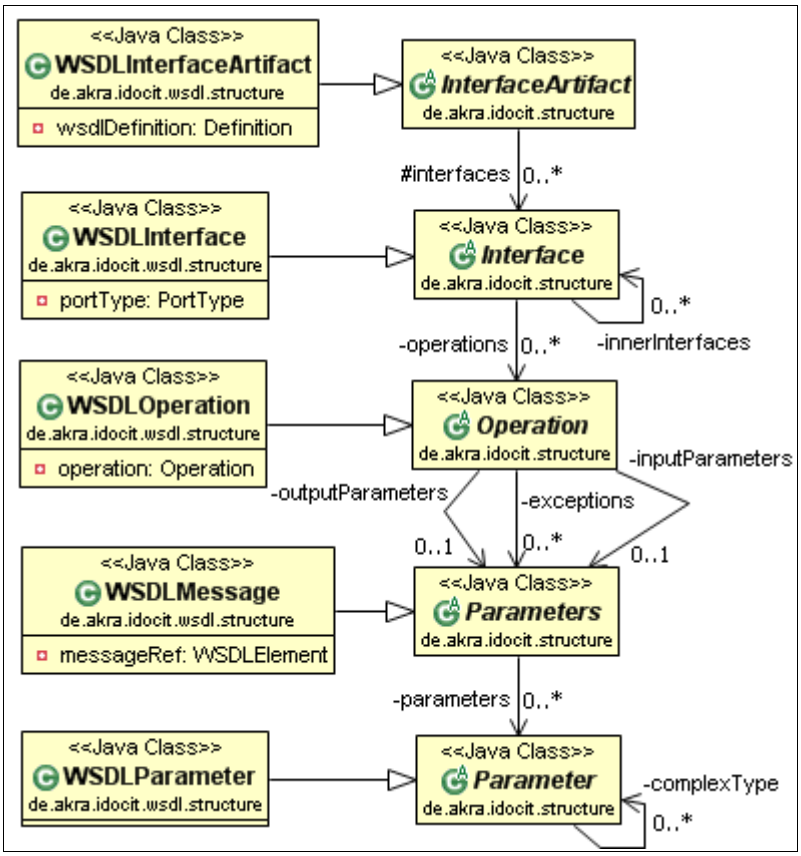


Abbildung 6.3: Klassen der allgemeinen Schnittstellenstruktur für WSDL

Die Klasse `WSDLParameter` benötigt kein zusätzliches Attribut, weil deren Dokumentation zu denen der `WSDLMessage` geschrieben wird. Eine `WSDLMessage` entspricht einem WSDL `<message>`-Element, sie hat aber eine Referenz auf ein `Input`, `Output` oder `Fault` Objekt einer `Operation` der WSDL4J-Objektstruktur. Das hat sie deswegen, weil ein `<message>`-Element i. d. R. außerhalb des `<portType>`-Elements definiert ist. Daher soll die Dokumentation der `WSDLMessage` und deren `WSDLParameter` nicht in das `<message>`-Element geschrieben werden, sondern in das entsprechende `<input>`-, `<output>`- oder `<fault>`-Element. Außerdem kann dasselbe `<message>`-Element in verschiedenen Operationen und somit auch in unterschiedlichen Kontexten verwendet werden. Die spezifischen Dokumentationen werden dann nach dem Lokalitätsprinzip zur Operation geschrieben. Die `WSDLOperation` hält eine Referenz auf das jeweilige

`Operation` Objekt aus der WSDL4J-Objektstruktur. Das Attribut `portType` der Klasse `WSDLInterface` referenziert ein `PortType` Objekt und aus der Klasse `WSDLInterfaceArtifact` wird das `Definition` Objekt referenziert.

Für die Elemente `<definitions>`, `<portType>` und `<operation>` wird die Dokumentation in das jeweilige enthaltene `<documentation>`-Element geschrieben. Für die `<input>`-, `<output>`- und `<fault>`-Elemente kann keine direkte Dokumentation erfasst werden, sondern für die enthaltenen `<message>`-Elemente, deren `<part>`-Elemente und ggf. für deren weiteren Elemente. Die Dokumentation dieser Elemente wird in das `<documentation>`-Element des jeweils dazugehörigen `<input>`-, `<output>`- oder `<fault>`-Elements geschrieben. Die Pfadangabe für das Attribut `signatureElement` fängt dabei mit dem Namen der `<message>` an.

Als Trennzeichen für den Attributpfad sollen ein Semikolon ';' als Elementtrenner, ein Plus '+' als Trennzeichen zwischen dem Datentyp und dem voll qualifizierten Bezeichner und ein Nummernzeichen '#' als Trennzeichen zwischen dem Bezeichner und der Namensraumbezeichnung verwendet werden. In dem Auszug aus der WSDL Spezifikation in Tabelle 6.6 ist zu sehen, dass Elementnamen den Wertebereich eines `NMTOKEN`'s aus der XML Schema Spezifikation haben.[56] Dieser Wertebereich schließt die gewählten Zeichen nicht mit ein, weswegen sie konfliktfrei eingesetzt werden können.[57]

56 Vgl. Biron, Paul V / Malhotra, Ashok: XML Schema Part 2, 2001, „NMTOKEN“
57 Vgl. Bray, Tim et al.: XML 1.0, 2000, „Nmtoken“

Beispiel:

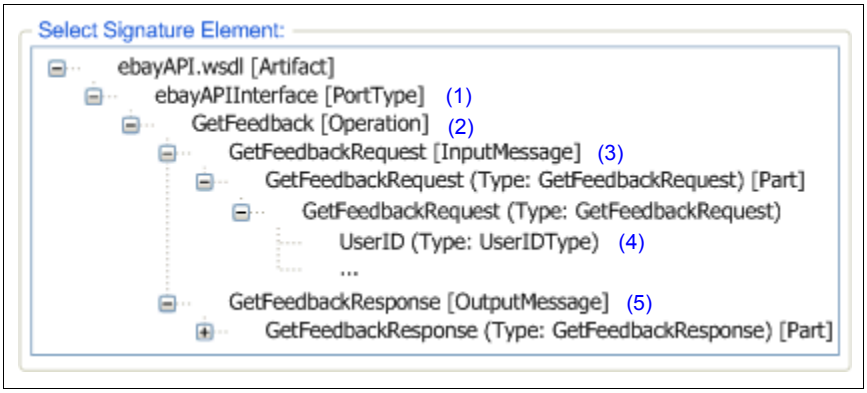


Abbildung 6.4: Anzeige der Struktur der Operation `GetFeedback` *aus der eBay Trading API im iDocIt! UI*

Abbildung 6.4 zeigt die Struktur der eBay Trading API für die Operationen `GetFeedback`, wie sie in der grafischen Oberfläche von iDocIt! angezeigt werden würde. Korrespondierend dazu zeigt die Tabelle 6.7, wohin iDocIt! die erfassten Dokumentationen in die WSDL schreibt. Die Dokumentation zum `PortType` (1) und zur `Operation` (2) werden jeweils direkt in das jeweilige Element geschrieben. Die Dokumentationsblöcke der `InputMessage` (3) und des Elements `UserID` (4) werden in das `<documentation>`-Element des `<input>`-Elements geschrieben. Die Dokumentation der `OutputMessage` (5) in das `<output>`-Element.

```
<wsdl:portType name="eBayAPIInterface">
  <wsdl:documentation>...</wsdl:documentation> (1)
  <wsdl:operation name="GetFeedback">
    <wsdl:documentation>
      <docpart role="ACTION" scope="EXPLICIT"> (2)
        <addressee group="Developer">...</addressee>
      </docpart>
      <docpart role="FORMULA" scope="IMPLICIT"> (2)
        <addressee group="Developer">...</addressee>
      </docpart>
    </wsdl:documentation>
    <wsdl:input message="GetFeedbackRequest">
      <wsdl:documentation>
        <docpart role="..."
         signatureElement="GetFeedbackRequest"> (3)
          <addressee group="Developer">...</addressee>
        </docpart>
        <docpart role="PRIMARY KEY"
         signatureElement="...;UserID+UserIDType"> (4)
          <addressee group="Developer">...</addressee>
        </docpart>
      </wsdl:documentation>
    </wsdl:input>
    <wsdl:output message="GetFeedbackResponse" />
      <wsdl:documentation>...</wsdl:documentation> (5)
  </wsdl:operation>
</wsdl:portType>
```

Tabelle 6.7: Zuordnung der iDocIt!-Dokumentationsblöcke in die WSDL der eBay Trading API

6.3 Parser für Java

Die Parser-Erweiterungen für Java ist unter der geforderten Prämisse entwickelt worden, dass die Quelldatei nur um die Dokumentation verändert wird. D. h. dass der vorhandene Inhalt bestehen bleiben soll, sodass der Programmcode nicht verändert wird. Unterschiedliche Formatierungen (z. B. Leerzeichen und Zeilenumbrüche) verändern bei Java nicht den Programmcode, da es beispielsweise in Blöcke und Anweisungen, die mit einem Semikolon beendet werden, aufgeteilt ist. Eine Veränderung des Formats ist demnach nicht entscheidend, aber sie sollte trotzdem bestmöglich vermieden werden. In z. B. einer Versionskontrolle sollen bei einem Dateivergleich nur die wichtigen Änderungen berücksichtigt werden. Unnötige Formatänderungen sollen diese nicht überlagern und auch nicht zu unnötig vielen Dateiänderungen führen.

Von der Parser-Erweiterung werden nur bestimmte Teile der Java Spezifikation Elementen aus der allgemeinen Schnittstellenstruktur zugeordnet. Die restlichen Teile müssen aber trotzdem eingelesen und behandelt werden, damit die Datei vollständig geschrieben werden kann. Um den Entwicklungs- und Implementierungsaufwand für einen vollständigen Parser für Java mit zurückschreiben der Daten zu vermeiden, wird auf einen existierenden Parser zurückgegriffen.

Zunächst werden einige existierende Parser evaluiert, ob deren Funktionalität für den Zweck Java mit Javadoc Kommentaren zu parsen und zu speichern ausreicht. Daran folgend wird die Struktur der Dokumentationen von iDocIt! entwickelt, wie sie sich in Javadoc integrieren soll. Zum Schluss werden einige Details zur Implementierung beschrieben.

6.3.1 Existierende Java Parser

Es existieren einige Parser für die Programmiersprache Java, die eine Java Objektstruktur aus einer Java-Quelldatei erstellen können. Für die Implementierung der Java Parser-Erweiterung wurden u. a. JaxMeJS (JaxMe JavaSource)[58], javapaser[59] und die Eclipse JDT (Java development tools)[60] näher evaluiert.

Die API des JaxMeJS Frameworks bietet alle nötigen Methoden, um eine Java-Quelldatei zu parsen, zu manipulieren und wieder zu speichern. Javadoc Kommentare können Elementen hinzugefügt werden und in diesen können sogar einzelne Javadoc Tags eingefügt werden. Aber leider kann der Parser von JaxMeJS keine Kommentare aus der Quelldatei einlesen. Daher ist er für iDocIt! unbrauchbar.

Die API des javaparsers hat einen gleichwertigen Funktionsumfang wie die von JaxMeJS. Aber im Gegensatz zu JaxMeJS kann er auch zu den Elementen einer Java-Quelle (z. B. Interfaces, Klassen, Methoden und Attribute) die Javadoc Kommentare einlesen. Aber außer das jeweils dazugehörige Javadoc werden keine weiteren Kommentare berücksichtigt. Beispielsweise würden keine normalen Kommentare, die zwischen Methoden geschrieben sind, berücksichtigt werden. Diese würden nach dem Speichern der Struktur gelöscht sein, was nach der eingangs geforderten Prämisse nicht passieren darf.

Die JDT werden auch in Eclipse für den Java-Editor eingesetzt. Sie parsen eine Java-Quelldatei vollständig. Jedes Element kann in dem entstandenen abstrakten Syntaxbaum (ASB) manipuliert oder entfernt werden, aber es können auch neue Elemente hinzugefügt werden. Zu den Elementen gehören auch Javadoc und normale Kommentare. Der ASB bietet auch die Möglichkeit an durch Reflection weitere Informationen über die einzelnen Elemente zu erhalten (z. B. Struktur/Attribute der Datentypen). Das Speichern des ASBs wird auf Befehl von den JDT übernommen.

58 Vgl. o.V.: JaxMeJS, o.J.
59 Vgl. o.V.: javaparser, o.J.
60 Vgl. o.V.: Java development tools (JDT) Core, o.J.

Die JDT bieten die beste Unterstützung für das Vorhaben der Bachelorarbeit, weswegen diese für die Parser-Erweiterung für Java gewählt wurde.

6.3.2 Struktur der Dokumentation in Javadoc

Die Dokumentationen von iDocIt! sollen die Dokumentation von Javadoc erweitern. Es soll also die Syntax und der Aufbau von Javadoc verwendet werden, aber der Inhalt der Beschreibungen, der bei Javadoc Freitext ist, soll strukturiert werden.

Eine Strukturierung wäre z. B. mit Javadoc Tags möglich, die frei definiert werden können. Dadurch würde die Dokumentation aber sehr wahrscheinlich zu überfüllt von Tags werden. Es müssten für jeden Dokumentationsblock Tags für den Elementpfad, den Sichtbarkeitsbereich, der thematischen Rolle und den Adressaten eingefügt werden.

Javadoc unterstützt auch HTML Formatierungen in den Beschreibungen. Daher soll auf dieses Format zurückgegriffen werden um die Dokumentationsblöcke darzustellen. Eigentlich sollte auf HTML Formatierungen in Kommentaren verzichtet werden, weil sie die Leserlichkeit der Kommentare im Quelltext verschlechtern können und durch die zusätzlich eingebrachte Beschreibungssprache die Verständlichkeit erschweren kann. Die Leser des Quelltextes müssen dann nicht nur die eigentliche Programmiersprache (Java) und die Dokumentationen lesen und verstehen, sondern auch noch HTML, was keine Informationen enthält, sondern nur Formatierungen.[61] Trotzdem sollen die Dokumentationsblöcke in HTML formatiert werden, weil sie dadurch einheitlich und übersichtlich dargestellt werden können (z. B. in der resultierenden API-Dokumentation im HTML Format) und sie einfach mit einem XML Parser wieder eingelesen werden können.

61 Vgl. Martin, Robert C.: Clean Code, 2009, S. 69

Ein Dokumentationsblock soll in einer HTML Tabelle mit zwei Spalten dargestellt werden. In der ersten Spalte stehen die Attributnamen und in der zweiten Spalte die dazugehörige Beschreibung. Die Reihenfolge der Attribute ist:

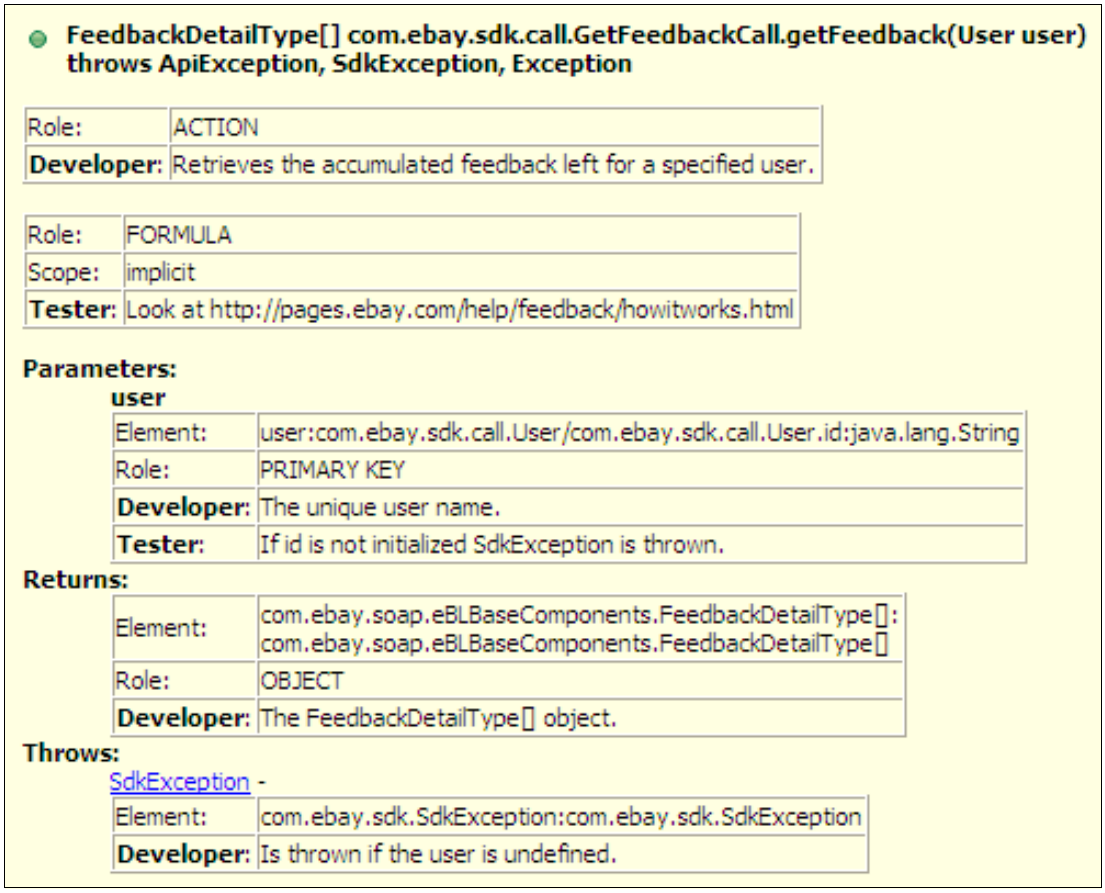
Element, Role, Scope, Namen von Adressaten.

Das Attribut Element wird nur bei Parametern, Ergebnistyp und Ausnahmen einer Methode verwendet. Wenn keine thematische Rolle für ein Element vergeben ist, wird das Attribut „Role“ auch nicht angezeigt. Wenn eine vergeben wurde, wird die thematische Rolle in der zweiten Spalte eingefügt. Bei dem Attribut „Scope“ bedeutet ein nicht Vorhandensein, dass es EXLPICIT ist, ansonsten ist es mit dem Wert IMPLICIT vorhanden. Diese Konvention wurde gewählt, um die Dokumentation nicht unnötig zu verlängern. Nach den fest vergebenen Attributen folgen die variablen Namen von Adressaten. Hierbei wird der Text in der ersten Spalte als Name für einen Adressaten verwendet und der Text in der zweiten Spalte als Freitextbeschreibung für diesen Adressaten.

Abbildung 6.5 zeigt diese Struktur anhand einer beispielhaften Javadoc Dokumentation im HTML Format. Dazu wurde die Methode `GetFeedback` aus der Klasse `GetFeedbackCall` des eBay SDK for Java[62] als Basis verwendet und leicht verändert, damit die Aspekte besser dargestellt werden können. Es wurde eine Klasse `User` mit dem Attribut `id` vom Typ `String` erstellt und als Eingabeparameter der Methode `GetFeedback` verwendet. Weiterhin ist nur eine der drei Ausnahmen der Methode exemplarisch aufgeführt.

Wie in dem Beispiel zu sehen ist, werden die Dokumentationsblöcke entsprechend den dokumentierten Elementen an den richtigen Stellen eingefügt. Beispielsweise werden Dokumentationsblöcke für eine generelle Beschreibung an den Anfang eines Javadoc Kommentars geschrieben (ohne ein Tag) und die Dokumentationsblöcke für Eingabeparameter zu dem jeweiligen Tag des Parameters. In dem Beispiel zu dem Tag `@param user`.

62 Vgl. o.V.: eBay SDK for Java Library Reference Documentation, o.J., „`GetFeedbackCall`“



FeedbackDetailType[] com.ebay.sdk.call.GetFeedbackCall.getFeedback(User user) throws ApiException, SdkException, Exception

Role:	ACTION
Developer:	Retrieves the accumulated feedback left for a specified user.

Role:	FORMULA
Scope:	implicit
Tester:	Look at http://pages.ebay.com/help/feedback/howitworks.html

Parameters:

user

Element:	user:com.ebay.sdk.call.User/com.ebay.sdk.call.User.id:java.lang.String
Role:	PRIMARY KEY
Developer:	The unique user name.
Tester:	If id is not initialized SdkException is thrown.

Returns:

Element:	com.ebay.soap.eBLBaseComponents.FeedbackDetailType[]: com.ebay.soap.eBLBaseComponents.FeedbackDetailType[]
Role:	OBJECT
Developer:	The FeedbackDetailType[] object.

Throws:

SdkException -

Element:	com.ebay.sdk.SdkException:com.ebay.sdk.SdkException
Developer:	Is thrown if the user is undefined.

Abbildung 6.5: Javadoc Beispiel in HTML mit iDocIt! Dokumentationsblöcken

6.3.3 Implementierungsdetails des Java-Parsers

Die Parser-Erweiterung für Java soll die Sprachspezifikation in der aktuellen dritten Edition[63] unterstützen. Bei der Entwicklung der allgemeinen Schnittstellenstruktur in Kapitel 5.1.2 wurden bereits die wichtigsten Schnittstellenelemente auf abstraktem Niveau vorgestellt. In Tabelle 6.8 ist die Spezifikation einer Interface Deklaration noch einmal auszugsweise in der EBNF aufgeführt, damit der Aufbau und die Zuordnung zu den Klassen der allgemeinen Schnittstellenstruktur gezeigt werden kann. Es sind nicht alle Elemente der Spezifikation in dem Auszug enthalten und die Produktionen werden teils nicht vollständig bis zum Terminalsymbol aufgeführt. Die groß geschriebenen Wörter (z. B. `InterfaceDeclaration` und `Identifier`)

63 Siehe Gosling, James et al.: Java Language Specification, 2005

stellen Nichtterminalsymbole und die klein geschriebenen Wörter (z. B. `class` und `interface`) sowie die Zeichen in einfachen Hochkommata (z. B. `'{'` und `';'`) stellen Terminalsymbole dar. Ein vertikaler Strich „|“ trennt alternative Ergebnisse einer Produktion, mit eckigen Klammern „`[...]`“ eingeschlossene Symbole sind optional (ein- oder keinmal vorhanden) und mit geschweiften Klammern eingeschlossene Symbole „`{...}`“ können null bis beliebig oft wiederholt werden.

```
InterfaceDeclaration = NormalInterfaceDeclaration

NormalInterfaceDeclaration = { InterfaceModifier } interface
                              Identifier [ ExtendsInterfaces ]
                              InterfaceBody

InterfaceBody = '{' { InterfaceMemberDeclaration } '}'

InterfaceMemberDeclaration = InterfaceDeclaration
                              | AbstractMethodDeclaration
                              | ClassDeclaration | ';'

ClassOrInterfaceType = ClassType | InterfaceType

AbstractMethodDeclaration = { AbstractMethodModifier } ResultType
                             MethodDeclarator [ Throws ] ';'

ClassDeclaration = NormalClassDeclaration | EnumDeclaration

NormalClassDeclaration = { ClassModifier } class Identifier
                          [ Super ] [ Interfaces ] ClassBody

ClassBody = '{' { ClassBodyDeclaration } '}'

ClassBodyDeclaration = ClassMemberDeclaration
                       | ConstructorDeclaration

ClassMemberDeclaration = MethodDeclaration | FieldDeclaration
                          | ClassDeclaration | InterfaceDeclaration
                          | ';'
```

Tabelle 6.8: Auszug der Java Sprachspezifikation für eine Interface Deklaration[64] in EBNF[65]

Wie dem Auszug der Interface Deklaration entnommen werden kann, können Schnittstellen (`NormalInterfaceDeclaration`) auch die vom Aufbau her sehr ähnlichen Klassen (`NormalClassDeclaration`) und Enumerationen (`EnumDeclaration`) enthalten. Klassen und Enumerationen müssen daher auch von

64 Vgl. Gosling, James et al.: Java Language Specification, 2005, S. 260ff
65 Vgl. Anhang C

der Parser-Erweiterung behandelt werden. Deswegen bedeutet es keinen nennenswerten Mehraufwand, wenn Klassen und Enumerationen auch direkt mit iDocIt! dokumentiert werden können, nicht nur als innere Deklaration.

```
InterfaceModifier      = public | protected | private | abstract
                         | static
AbstractMethodModifier = public | abstract
ClassModifier          = public | protected | private | abstract
                         | static | final
MethodModifier         = public | protected | private | abstract
                         | static | final | synchronized
```

Tabelle 6.9: Auszug der möglichen `Modifier` der Java Sprachspezifikation in EBNF[66]

Bei der Dokumentation von Parametern einer Operation soll sich auf die nach der JavaBeans Spezifikation öffentlich zugänglichen Attributen beschränkt werden. Andere Attribute sollen nicht berücksichtigt werden. D. h. es sollen nur Attribute berücksichtigt werden, die einen `public` Getter (Lesezugriff) oder Setter (Schreibzugriff) nach dem JavaBeans Design Muster haben. Nach diesem Muster muss der Getter für ein einfaches Attribut mit `get` beginnen und der Name des Attributs folgen, wobei der erste Buchstabe groß geschrieben wird.[67,68]

```
public class EMailAccount {
  public boolean enabled;
  public boolean freeAccount;
  private int id;
  private User user;

  public boolean isFreeAccount() { return freeAccount; }
  public void setFreeAccount(boolean free) { freeAccount = free; }
  public int getId() { return id; }
  public int getUser() { return user; }
  public void setUser(User u) { user = u; }
}
```

Tabelle 6.10: Beispiel einer Java-Klasse zur Erläuterung der JavaBeans Spezifikation

66 Vgl. Anhang C
67 Vgl. Hamilton, Graham: JavaBeans, 1997, „Accessor methods“, S. 40
68 Vgl. Hamilton, Graham: JavaBeans, 1997, „Simple properties“, „Boolean properties“, S. 55

In Tabelle 6.10 ist dieses exemplarisch für die Attribute `id` und `user` dargestellt. Bei einem Attribut vom Typ `boolean` fängt der Getter mit `is` an, gefolgt von dem groß geschriebenen Bezeichner (`isFreeAccount()`). Die Setter folgen dem gleichen Muster, fangen nur immer mit `set` an. In dem Beispiel in Tabelle 6.10 sind die Attribute `id`, `user` und `freeAccount` nach der JavaBeans Spezifikation nach außen sichtbar und würden vom Parser berücksichtigt werden. Das `public` Attribut `enabled` ist zwar nach der Java Spezifikation nach außen sichtbar, wird aber nicht berücksichtigt, weil es keinen Getter und/oder Setter nach der JavaBeans Spezifikation hat.

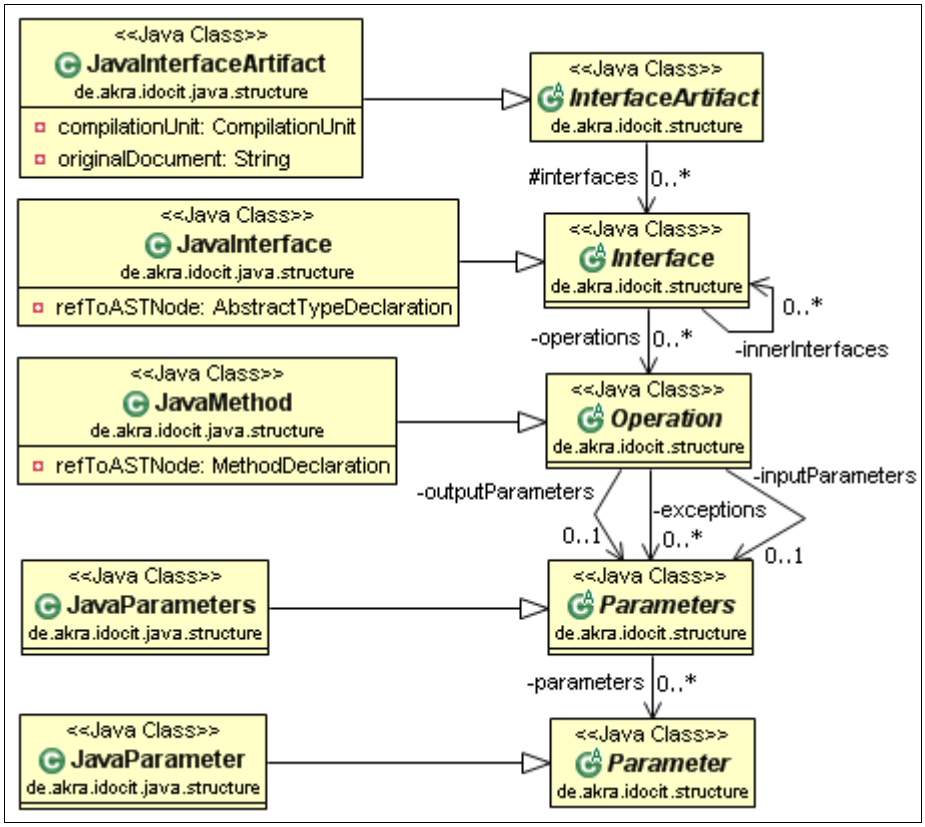


Abbildung 6.6: Klassen der allgemeinen Schnittstellenstruktur für Java

Der ASB, der von den JDT aus einer Java-Quelldatei erstellt wird, ist der Ausgangspunkt, um die allgemeine Schnittstellenstruktur eines `JavaInterface-Artifact`'s aufzubauen (siehe Abbildung 6.6). Die erweiterten Klassen

`JavaInterfaceArtifact`, `JavaInterface` und `JavaMethod` haben jeweils zusätzliche Attribute erhalten, damit die Änderungen an den Dokumentationen an die richtigen Stellen des ASBs geschrieben werden können.

In der Klasse `JavaInterfaceArtifact` hält das Attribut `compilationUnit` eine Referenz zur Repräsentation der geparsten Datei als ASB und das Attribut `originalDocument` enthält die `String` Repräsentation des unveränderten Inhalts der geparsten Datei. Das zweite Attribut wird beim Speichern benötigt, denn die `CompilationUnit` merkt sich nur die Änderungen, die daran gemacht werden. Vor dem Zurückschreiben in die Datei werden die Änderungen in den Originalinhalt eingepflegt und dann alles in die Datei geschrieben. Die Klassen `JavaInterface` und `JavaMethod` haben jeweils eine Referenz auf das entsprechende `AbstractTypeDeclaration` (z. B. `InterfaceDeclaration` oder `Class-Declaration`) und `MethodDeclaration`.

Die Klassen `JavaParameters` und `JavaParameter` haben keine weiteren Attribute erhalten, weil zu `JavaParameters` keine Dokumentation erfasst werden kann und die Dokumentation von `JavaParameter` wird mit in die der `JavaMethod` integriert. Die Klasse `JavaParameters` ist hier nur ein Container für eine Menge von `JavaParameter`, sie hat kein korrespondierendes Element in der Java Sprachspezifikation.

In der Java Parser-Erweiterung kommt zum ersten Mal das Attribut `documentationAllowed` vom `SignatureElement` (siehe Abbildung 5.5, S. 47) zum Einsatz. Über dieses Attribut wird gesteuert, ob für ein `SignatureElement` eine Dokumentation in dem UI erstellt werden kann. Die Klassen `JavaInterfaceArtifact` und `JavaParameters` haben keine korrespondierenden Elemente in der Java Sprachspezifikation. Daher ist für diese Elemente auch kein Platz für eine Dokumentation als Javadoc in einer Java-Quelldatei vorgesehen. Es macht dann keinen Sinn für diese Elemente Dokumentationen in dem UI zu erfassen.

Wie bereits erwähnt wird die Dokumentation der `JavaParameter` mit in die der zugehörigen Methode integriert. Dazu müssen die einzelnen Dokumentationen wieder einen eindeutigen Pfad zur Identifikation des Elements enthalten. Der Pfad beginnt aber nicht mit der Klasse `JavaParameters`, wie bei der WSDL, sondern mit der Klasse `Parameter`.

Als Trennzeichen für den Attributpfad sollen ein Schrägstrich '/' als Elementtrenner, ein Doppelpunkt ':' als Trennzeichen zwischen dem Datentyp und dem voll qualifizierten Bezeichner und ein Punkt '.' als Trennzeichen zwischen dem Bezeichner und der Namensraumbezeichnung (Java Paket) verwendet werden. Alle drei Zeichen sind laut Spezifikation in einem Bezeichner nicht erlaubt.[69] Der Punkt als Namensraumtrenner wird dafür bereits von Java verwendet. Von iDocIt! wird dieses nach der Definition gehandhabt, dass in einem qualifizierten Bezeichner der Bezeichner nach dem letzten Punkt der einfache Name des Elements ist.

Beim Einlesen eines Javadoc Kommentarblocks soll der Prototyp von iDocIt! erst einmal nur Beschreibungen berücksichtigen, die im iDocIt! Format vorliegen. Andere Beschreibungen werden ignoriert und beim erneuten Speichern der Dokumentation überschrieben. In einer folgenden Version von iDocIt! sollen bestehende Dokumentation eingelesen werden und in die Struktur der iDocIt! Dokumentation überführt werden.

69 Vgl. Gosling, James et al.: Java Language Specification, 2005, „Identifier“, S. 19ff

6.4 User Interface (UI)

In diesem Kapitel wird das Zusammenspiel der UI Klassen und deren Realisierung näher beschrieben.

In Kapitel 5.4 wurde bereits erwähnt, dass die Verwaltung der UI Klassen in der Klasse `EditArtifactDocumentationComposite` geschieht. Durch die Nutzung des AKRA GUI Frameworks „POC-UI“ haben alle UI Klassen die gleichen Schnittstellen, was das Arbeiten mit ihnen erheblich erleichtert.[70]

Wie in Abbildung 6.7 zu sehen ist, erben alle Klassen von der abstrakten POC-UI Klasse `AbsComposite` die einheitliche Schnittstelle. In der Methode `initGUI` werden die grafischen Komponenten der Klasse erstellt und soweit initialisiert, wie es ohne einen definierten Zustand möglich ist. In der Methode `initListener` werden die benötigten Listener erzeugt, um auf Ereignisse reagieren zu können, die von den anderen Komponenten gefeuert werden. Über die Methode `fireChangeEvent` wird ein Ereignis ausgelöst, dass den lauschenden Komponenten mitteilt, dass diese Komponente verändert worden ist und sich die anderen Komponenten ggf. auch ändern sollten. Mit der Methode `addAllListener` werden die initialisierten Listener den jeweiligen Komponenten hinzugefügt und mit `removeAllListener` wieder entfernt. In der Methode `doCleanUp` werden alle Ressourcen freigegeben, die nicht automatisch von z. B. dem Java Garbage Collector freigegeben werden.

70 Vgl. Krause, Jan Christian / Joneleit, Kai Benjamin: Prozessorientierte Konstruktion von Benutzeroberflächen, 2009

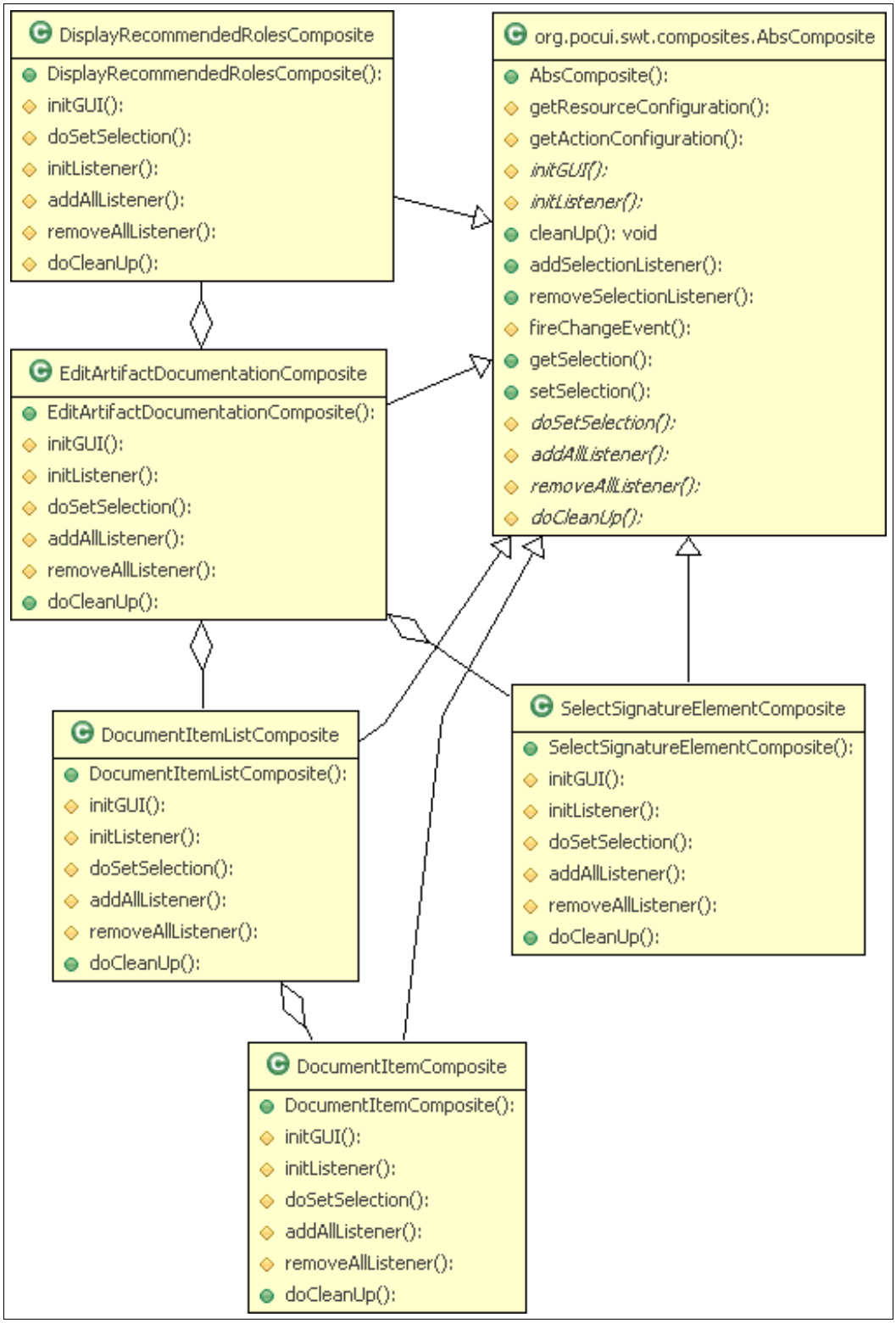


Abbildung 6.7: UI Klassen mit Methodennamen (keine vollständige Methodensignatur)

Über die Methode `setSelection` bzw. `doSetSelection` wird der Zustand der jeweiligen Komponente gesetzt. Diese Methode erhält als Eingabe ein Objekt, das alle Rohdaten beinhaltet, die von der Komponente selbst und deren abhängigen Komponenten benötigt werden. Aus diesem `Selection`-Objekt wird der vollständige Zustand der Komponenten hergestellt.

Abbildung 6.8 zeigt als Beispiel einen Auszug der Klasse `DocumentItemList-CompositeSelection`. Sie enthält die Daten für den Zustand der Klasse `DocumentItemListComposite` und deren enthaltenen `DocumentItem-Composites`. Es enthält u. a. folgende Attribute:

- `documentations`: Liste der anzuzeigenden Dokumentationsblöcke
- `addresseeList`: Liste der für iDocIt! konfigurierten Adressaten
- `thematicRoleList`: Liste der für iDocIt! konfigurierten thematischen Rolle
- `documentationAllowed`: Flag, ob eine Dokumentation für das selektierte Element erlaubt ist.

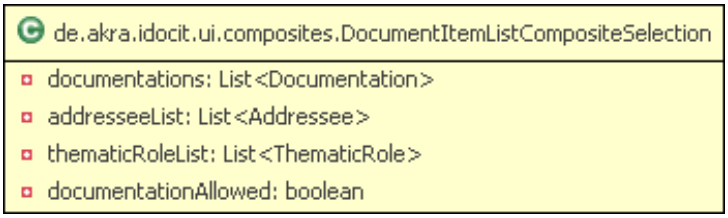


Abbildung 6.8: Auszug der Klasse `DocumentItemListCompositeSelection`

Das `Selection`-Objekt beinhaltet immer den aktuellen Zustand einer Komponente. Es wird in der jeweiligen Komponente zwischengespeichert und wenn eine Änderung eintritt, wird es geändert und das `ChangeEvent` (`fireChangeEvent`) ausgelöst. Die Komponenten, die einen Listener bei der feuernden Komponente registriert haben, rufen dann das geändert `Selection`-Objekt ab und ändern danach entsprechend selbst ihren Zustand und den der abhängigen Komponenten. Danach wird wieder `ChangeEvent` (`fireChangeEvent`) gefeuert, damit die „darüber liegenden Komponenten“ über die Änderung informiert werden usw.

Dieses Prinzip soll beispielhaft an den Komponenten `DocumentItemList-Composite` und `DocumentItemComposite` gezeigt werden. Das `Selection`-Objekt des `DocumentItemListComposites` enthält u. a. eine Liste von `Documentation` Objekten, die in dem `DocumentItemListComposite` dargestellt werden sollen. Das `DocumentItemListComposite` selbst besitzt eine Liste von `DocumentItemComposites` und eine Liste von Schaltflächen, über die ein `DocumentItemListComposite` aus der Liste entfernt werden kann, sowie eine eine Schaltfläche zum Hinzufügen weiterer Dokumentationen. Die beiden genannten Listen sind immer synchron gehalten. Für jedes `DocumentItemComposite` gibt es an der gleichen Stelle der Liste eine Schaltfläche, über die das `Composite` entfernt werden kann.

Wenn auf die Schaltfläche zum Hinzufügen einer Dokumentation geklickt wird, wird ein neues `Documentation` Objekt erstellt, dieses wird der `Documentation` Liste im `Selection`-Objekt hinzugefügt und die Methode `fireChangeEvent` aufgerufen. Das `EditArtifactDocumentationComposite` registriert dann die Änderung und aktualisiert die Zustände der abhängigen Komponenten dementsprechend. Es setzt jeweils ein aktualisiertes `Selection`-Objekt über die Methode `setSelection`. Dadurch verändert sich dann der Zustand der Komponenten. Die Veränderung der Oberfläche wird also nicht direkt von einer Komponente gemacht, sondern sie ändert lediglich ihr `Selection`-Objekt und informiert dann die Elternkomponente.

Da nun ein `Documentation` Objekt in der Liste des `Selection`-Objekts vorhanden ist, wird dafür ein `DocumentItemComposite` und eine Entfernen-Schaltfläche erzeugt und in die jeweilige Liste im `DocumentItemListComposite` hinzugefügt. Für das `DocumentItemComposite` wird das entsprechende `Selection`-Objekt erzeugt und mit den nötigen Daten initialisiert. Das `Selection`-Objekt wird dann über die `setSelection` Methode der

`DocumentItemComposite` übergeben, wodurch es ihren Zustand erhält (siehe Tabelle 6.11).

```
private void addNewDocumentItemComposite(
  Documentation documentation,         // documentation to display
  List<Addressee> addresseeList,       // list of all Addressees
  List<ThematicRole> thematicRoleList) // list of all ThematicRoles
{
  DocumentItemCompositeSelection selection =
      new DocumentItemCompositeSelection();
  selection.setDocumentation(documentation);
  selection.setAddresseeList(addresseeList);
  selection.setThematicRoleList(thematicRoleList);

  DocumentItemComposite docItemComp =
      new DocumentItemComposite(this.rootComposite, SWT.NONE);
  docItemComp.setSelection(selection);

  this.documentItemCompositeList.add(docItemComp);

  Button deleteButton = new Button(this.rootComposite, SWT.PUSH);
  deleteButton.setText("X");
  this.documentItemCompositeDeleteButtonList.add(deleteButton);
}
```

Tabelle 6.11: Auszug der Methode aus der Klasse `DocumentItemListComposite` zum Hinzufügen einer `Documentation` (`DocumentItemComposite`)

Wenn auf eines der Entfernen-Schaltflächen zu einem `DocumentItemComposite` geklickt wird, wird der Index der Schaltfläche in der Liste bestimmt und danach das `Documentation` mit dem gleichen Index aus der Liste des `Selection`-Objekts entfernt (siehe Tabelle 6.12). Danach wird die Änderung wieder über `fireChangeEvent` bekannt gemacht.

```
private void documentationItemCompositeDeleteButtonClicked(
  SelectionEvent e)
{
  // get index of the clicked delete button
  int row = this.documentItemCompositeDeleteButtonList
                  .indexOf(e.widget);

  // get current selection object
  DocumentItemListCompositeSelection listSelection =
     getSelection();

  // delete documentation out of the selection object
  listSelection.getDocumentations().remove(row);

  fireChangeEvent();
}
```

Tabelle 6.12: Auszug der Methode aus der Klasse `DocumentItemListComposite` zum Entfernen einer `Documentation`

7 Systembewertung

Nachdem der Prototyp des Werkzeugs iDocIt! entwickelt ist, soll dieses nun bewertet werden. Zuerst muss es sich in einem fiktiven Fallbeispiel bewähren. Ein Teil der Klasse `GetFeedbackCall` aus dem eBay SDK for Java[71] soll einmal mit den Bordmitteln von Eclipse mit Javadoc dokumentiert werden und einmal mit iDocIt!. Für dieses Fallbeispiel wird Eclipse mit Javadoc verwendet und keins der in Kapitel 4 evaluierten Werkzeuge, weil keins der Werkzeuge, das Java Dokumentieren kann, mehr als die sichtbaren Schnittstellenelemente einem Autor zur Dokumentation vorgibt. Eclipse hat demnach die selbe Funktionalität, aber den Vorteil, dass nicht noch ein Werkzeug detailliert beschrieben werden braucht.

Nach dem Fallbeispiel wird überprüft, ob iDocIt! alle in Kapitel 3 hergeleiteten Anforderungen an ein Dokumentationssystem erfüllt. Abschließend wird ein Fazit gezogen.

7.1 Fallbeispiel

In dem Fallbeispiel soll die Klasse `GetFeedbackCall` aus dem eBay SDK for Java (siehe Abbildung 7.1) soweit dokumentiert werden, dass ein Entwickler erfährt, wie das Feedback für einen bestimmten Benutzer abgerufen wird. Mögliche auftretende Ausnahmen sollen hier zur Vereinfachung nicht berücksichtigt werden. Dazu soll einmal das Javadoc mit iDocIt! und einmal mit den Hilfsmitteln von Eclipse (Raster von den sichtbaren Schnittstellenelementen) erstellt werden. Das Hauptaugenmerk liegt beim Dokumentieren auch darauf, wie gut ein Autor beim Dokumentieren unterstützt wird. Als Basis für die einzelnen Beschreibungen soll das original Javadoc der eBay SDK for Java API[72] dienen.

71 Vgl. o.V.: eBay SDK for Java, o.J.
72 Vgl. o.V.: eBay SDK for Java Library Reference Documentation, o.J., „`GetFeedbackCall`“

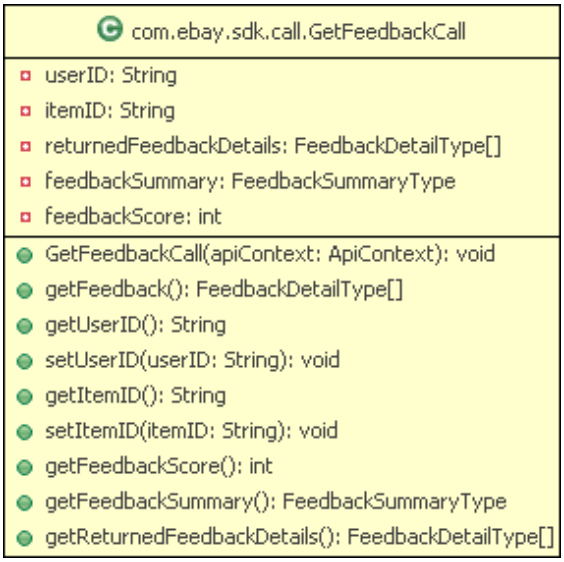


Abbildung 7.1: Auszug der Klasse GetFeedbackCall

7.1.1 Dokumentation mit iDocIt!

Bevor mit dem Dokumentieren angefangen werden kann, müssen für iDocIt! die thematischen Raster für die verwendeten Verben erstellt werden. In der Klasse werden nur zwei verschiedene Verben verwendet, *get* und *set*. Das mag erst einmal für das Erstellen der Raster positiv erscheinen, birgt aber Schwierigkeiten für das Dokumentieren, auf die später eingegangen wird. Zuerst sollen die thematischen Raster definiert werden.

In Tabelle 7.1 sind die thematischen Raster aufgeführt, die für das Fallbeispiel verwendet werden sollen. Da das Verb *get* in verschiedenen Kontexten verwendbar ist, wird es auch verschiedenen thematischen Rastern (Verbklassen) zugeordnet, mit unterschiedlichen thematischen Rollen. Das Verb *set* ist dagegen nur einem Raster zuordenbar. Damit der Kontext jedes Rasters besser verständlich ist, bekommen sie Namen (z. B. „Calculating Operations“) und z. T. sind weitere Verben aufgeführt.

Verbklasse / Thematisches Raster	Verben	Thematische Rollen
Setter	set	AGENT, ACTION, OBJECT, SOURCE
Getter	get	AGENT, ACTION, OBJECT, SOURCE
Calculating Operations	compute, calculate, get	AGENT, ACTION, OBJECT, SOURCE, FORMULA
Searching Operations	search, find, seek, get	AGENT, ACTION, OBJECT, SOURCE, CRITERION, PRIMARY KEY

Tabelle 7.1: Definierte thematische Raster für das Fallbeispiel

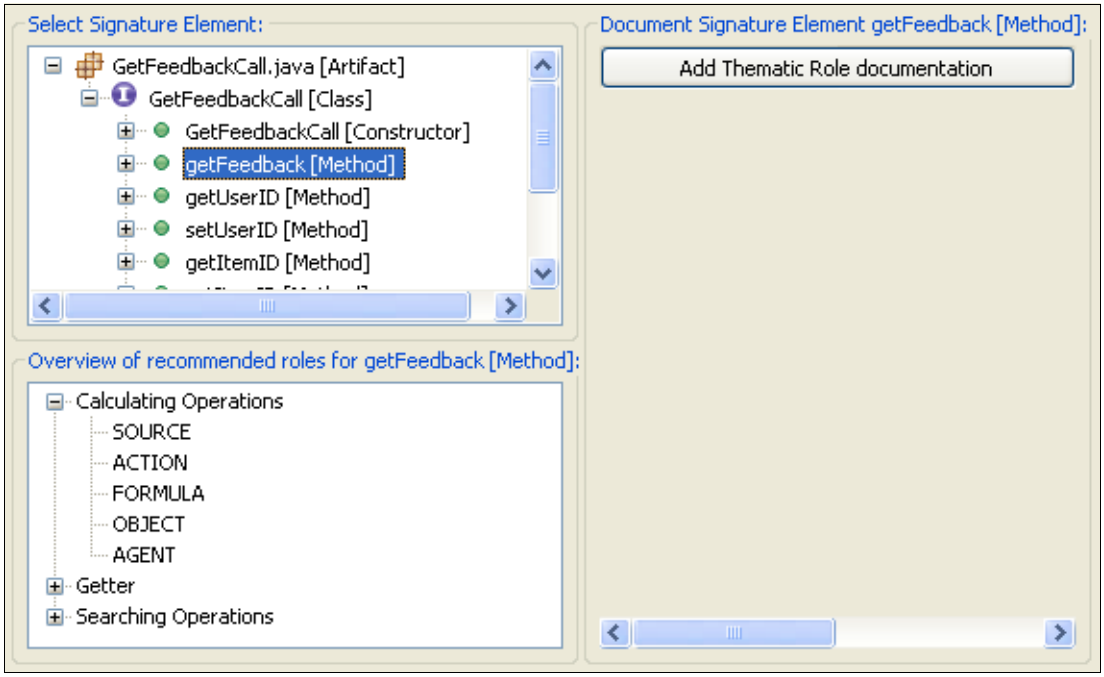


Abbildung 7.2: Screenshot der Klassenstruktur von `GetFeedbackCall` in iDocIt!

Die Vorbereitungen sind nun abgeschlossen und es kann begonnen werden die Klasse in iDocIt! zu dokumentieren. Über das Kontextmenü der Datei wird iDocIt! für die Klasse `GetFeedbackCall` gestartet (siehe Abbildung 7.2). Das Werkzeug muss immer in einem separaten Editor geöffnet werden. Änderungen in der Quelldatei werden dann nicht mehr berücksichtigt. Man muss also immer den Ablauf verfolgen, dass implementiert wird, dann iDocIt! geöffnet wird, Dokumentation erfasst und gespeichert wird und dann iDocIt! wieder geschlossen wird. Danach kann weiter

implementiert werden. Dieser Vorgang ist für eine schnelle Dokumentation, wie sie direkt mit Javadoc möglich ist (siehe Kapitel 7.1.2), umständlich.

Mit der Methode `getFeedback()` wird der API Aufruf zum eBay Webservice getätigt und das Feedback abgerufen. In dem geöffneten iDocIt! Editor werden die empfohlenen thematischen Rollen für die Methode angezeigt, wenn sie selektiert wird. Da das Verb *get* drei thematischen Rastern zugeordnet ist (siehe Tabelle 7.1), werden diese drei in der Oberfläche (Abbildung 7.2 im Bereich unten links) als Vorschlag von zu dokumentierenden Elementen angezeigt.

Hier zeigt sich die erwähnte Schwierigkeit. Da *get* ein unpräzises Verb ist, kann es in vielen Kontexten verwendet werden. Der Autor muss dann entscheiden, um welchen Kontext es sich handelt und dementsprechend ein Raster wählen, an das er sich beim Dokumentieren hält.

In diesem speziellen Fall ist die Entscheidung für ein Raster schwierig, denn die Methode kann zu den Kontexten aller drei thematischen Raster zugeordnet werden. Je nach dem Standpunkt kann man sagen, dass nur ein Wert abgerufen wird, also würde das „Getter“ Raster passen. Man könnte auch sagen, dass das Feedback Resultat vieler Einzelbestandteile ist. Es wird also mit einer definierten Formel berechnet. Dann würde das „Calculating Operations“ Raster passen. Oder man nimmt an, dass das Feedback für einen Benutzer oder Artikel (= Item) usw. gesucht wird (z. B. aus einer Datenbank). Dafür würde das „Searching Operations“ Raster passen. In so einem speziellen Fall wäre ein Mix von den Rastern möglich, denn die thematischen Raster bieten ein Standard Portfolio an thematischen Rollen an, die in der Regel für ein Verb dokumentiert werden sollten.

Die Methode `getFeedback()` ruft nicht nur einen einfachen Wert ab, sondern einen aus verschiedenen Bestandteilen und Bedingungen berechneten Wert. Außerdem macht sie dieses für einen bestimmten Benutzer oder Artikel. Es soll daher hauptsächlich nach dem Raster „Calculating Operations“ dokumentiert werden, aber

auch die Rolle PRIMARY KEY aus dem Raster „Searching Operations“ soll berücksichtigt werden.

Beim Klick in iDocIt! auf die Methode `getFeedback()` werden also die thematischen Raster angezeigt und diese sollen den Elementen der Methode zugeordnet werden und auch beschrieben werden.

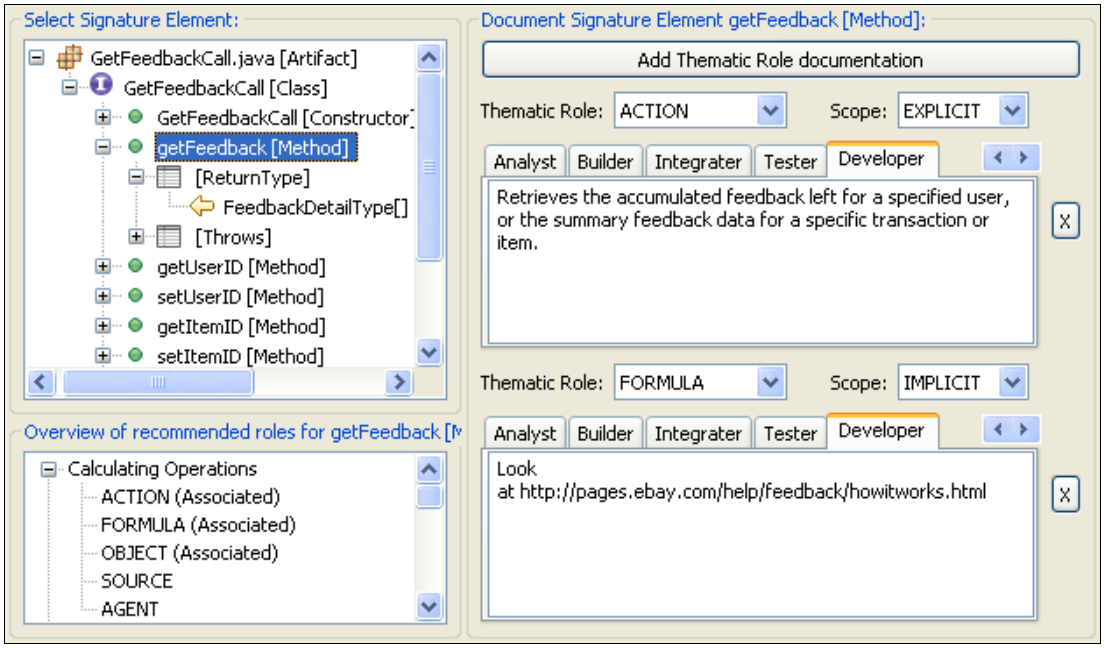


Abbildung 7.3: In iDocIt! dokumentierte Methode `GetFeedback()` der Klasse `GetFeedbackCall`

Als erstes können die thematischen Rollen den sichtbaren Elementen der Methode zugeordnet werden (siehe Abbildung 7.3). Die Rolle ACTION erhält die Methode, zusammen mit einer (allgemeinen) Beschreibung ihrer Aktion. Danach wird die Rolle OBJECT dem Ergebnistypen zugeordnet und das Ergebnis beschrieben.

Nun sind die nicht an der Methode sichtbaren Elemente dran. Dazu zählt die zugrunde liegende Formel (FORMULA), die Quelle der Daten (SOURCE), wer die ausführende Instanz (AGENT) der Methode ist und das Kriterium (PRIMARY KEY), das entscheidet für wen oder was das Feedback abgerufen werden soll.

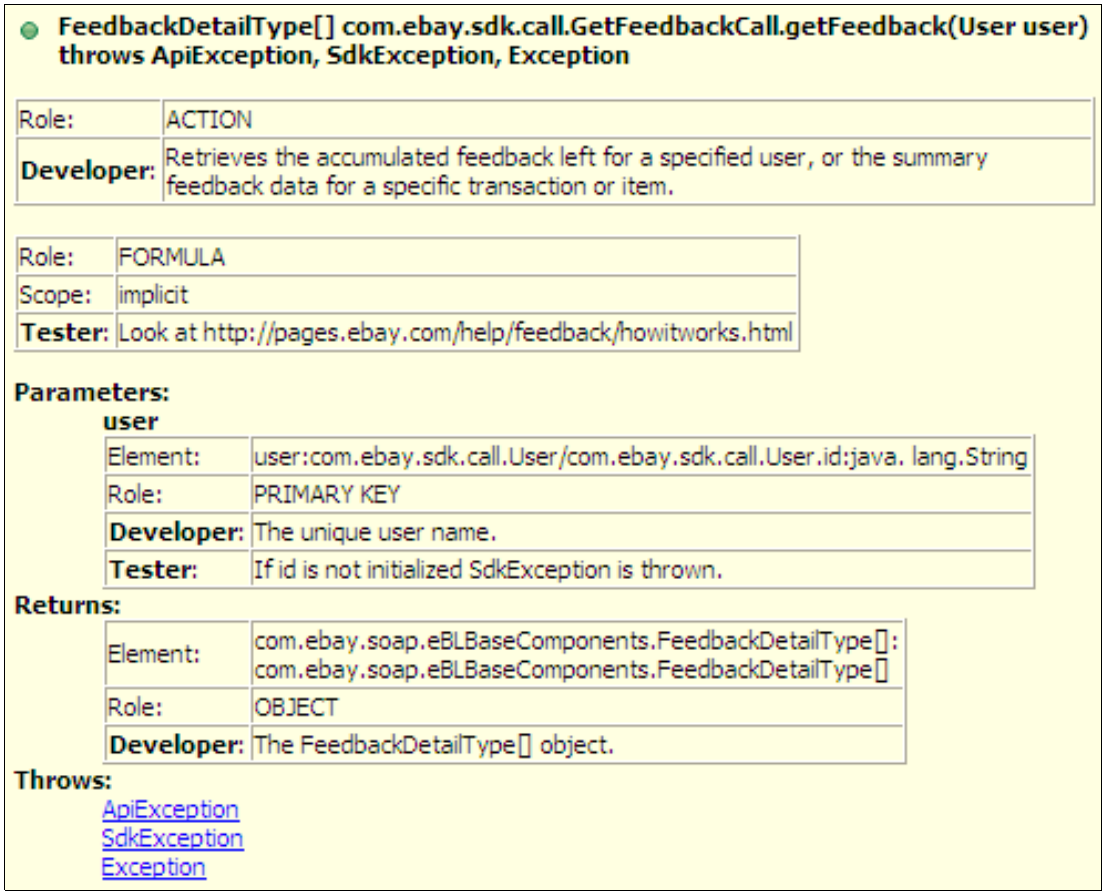
Die Rolle FORMULA wird der Methode zugeordnet und dort wird die Berechnungsvorschrift auch beschrieben. Die Rolle PRIMARY KEY kann keinem

Element der Methode zugeordnet werden, da diese Einstellung woanders geschieht. Auch die Rolle AGENT kann nicht der Methode zugeordnet werden, da sie sich nicht selbst ausführt. In der objektorientierten Welt führt das Objekt der Methode diese aus. Die Rolle AGENT wird demnach der Klasse `GetFeedbackCall` zugeordnet. Die Rolle SOURCE kann der Methode zugeordnet werden, aber da keine z. B. Datenbankverbindung oder Datei als Eingabeparameter in die Methode hineingeht, wird die Datenquelle auf Klassenebene verwaltet. Deswegen wird die Rolle SOURCE auch der Klasse `GetFeedbackCall` zugeordnet und dort beschrieben.

Nun braucht nur noch die Rolle PRIMARY KEY zugeordnet werden. Es muss also eine Methode existieren, mit der gesetzt wird, für wen das Feedback abgerufen werden soll. Nach kurzem Suchen ist die Methode `setUserID(String userID)` ermittelt. Wenn hierüber eine Benutzer ID gesetzt wird, wird beim Aufruf von `getFeedback()` dieses Kriterium berücksichtigt. Der Parameter `userID` erhält also die Rolle PRIMARY KEY und die Methode die Rolle ACTION zusammen mit einer kurzen Beschreibung ihrer Funktion.

Zum Schluss kann die erfasste Dokumentation gespeichert werden, wodurch sie in den Quelltext geschrieben wird. Wenn ein Entwickler diese Dokumentation liest, erfährt er genügend über die Funktionsweise, damit er das Feedback für einen Benutzer abrufen kann.

In Abbildung 7.4 ist die resultierende Dokumentation gezeigt, die direkt an der Methode `GetFeedback` erfasst wurde. Wie eingangs beschrieben, sind für die Ausnahmen keine Dokumentationen erfasst worden.



FeedbackDetailType[] com.ebay.sdk.call.GetFeedbackCall.getFeedback(User user) throws ApiException, SdkException, Exception

Role:	ACTION
Developer:	Retrieves the accumulated feedback left for a specified user, or the summary feedback data for a specific transaction or item.

Role:	FORMULA
Scope:	implicit
Tester:	Look at http://pages.ebay.com/help/feedback/howitworks.html

Parameters:

user

Element:	user:com.ebay.sdk.call.User/com.ebay.sdk.call.User.id:java. lang.String
Role:	PRIMARY KEY
Developer:	The unique user name.
Tester:	If id is not initialized SdkException is thrown.

Returns:

Element:	com.ebay.soap.eBLBaseComponents.FeedbackDetailType[]: com.ebay.soap.eBLBaseComponents.FeedbackDetailType[]
Role:	OBJECT
Developer:	The FeedbackDetailType[] object.

Throws:

ApiException
SdkException
Exception

Abbildung 7.4: HTML-Ansicht der iDocIt! Dokumentation für die Methode `GetFeedback`

7.1.2 Dokumentation mit Eclipse und Javadoc

Für die Dokumentation mit Eclipse sind keine Vorbereitungen nötig. Die Dokumentation wird auch direkt in den Quelltext geschrieben. Das Javadoc-Raster wird erstellt, indem über der Methode `getFeedback()` mit `/**` ein Kommentarblock angefangen wird und dann die Enter-Taste gedrückt wird. Dann wird das Raster mit den sichtbaren Elementen der Methodensignatur erstellt (siehe Tabelle 7.2).

```
/**
 *
 * @return
 */
public FeedbackDetailType[] getFeedback() { ... }
```

Tabelle 7.2: Javadoc-Raster für `getFeedback()`

Das Raster schlägt zur Dokumentation nur den Ergebnistypen (`@return`) und eine allgemeine Freitextbeschreibung vor (Leerzeile am Anfang des Kommentars). Nach der Beschreibung beider Elemente ist die Methode mit den notwendigen Tags nach der Javadoc Konvention beschrieben (siehe Tabelle 7.3).

```
/**
 * Retrieves the accumulated feedback left for a specified user, or
 * the summary feedback data for a specific transaction or item.
 *
 * @return The FeedbackDetailType[] object.
 */
public FeedbackDetailType[] getFeedback() { ... }
```

Tabelle 7.3: Ausgefülltes Javadoc-Raster für `getFeedback()`

Ein Entwickler könnte mit dieser Beschreibung das Feedback für einen Benutzer nicht abrufen. Es fehlen viele nötige Informationen, wie die Einstellungen von der Datenquelle, aber auch Konfiguration der Klasse bzw. für den Aufruf der Methode. Der Autor muss sich selbst erarbeiten, was alles benötigt wird, um mit der Klasse bzw. Methode zu arbeiten. Dieses muss er dann im Freitext beschreiben.

7.2 Realisierung der Anforderungen

Wie u. a. in dem vorherigen Kapitel gesehen werden kann, realisiert iDocIt! fast vollständig die allgemeinen Anforderungen an eine Schnittstellendokumentation aus Kapitel 3.1. Folgendes kann in iDocIt! dokumentiert werden.

- Eingaben
- Ausgaben
- Ausnahmen / Fehler
- Konfigurationsmöglichkeiten (z. B. von Datenbankverbindungen oder die Verwendung unterschiedlicher Algorithmen).

Das Werkzeug unterstützt aber nicht direkt das Referenzieren (Verlinkung) der Dokumentation von Datentypen oder Metadaten wie z. B. eine Versionsnummer. Da sich iDocIt! aber hauptsächlich in eine bestehende Dokumentationsmöglichkeit integriert, wie z. B. bei Javadoc, können weiterhin deren Möglichkeiten zur Dokumentation verwendet werden. Die weiteren Metadaten könnten daher über das Freitextfeld im iDocIt! UI mit erfasst werden.

Die Anforderung aus Kapitel 3.2, dass ein Dokumentationsformat thematische Rollen und adressatenspezifische Dokumentationen abbilden können muss, ist mit iDocIt! auch umgesetzt. Für jede unterstützte Programmier- oder Beschreibungssprache (WSDL und Java) wurde ein Format entsprechend den Möglichkeiten der Sprache entwickelt, dass die Anforderung erfüllt.

Auch an das Dokumentationswerkzeug wurden einige Anforderungen gestellt (siehe Kapitel 3.3). Es sollte

- eine grafische Oberfläche besitzen,
- thematische Raster aus einem Operationsbezeichner ableiten können,
- die thematischen Raster anzeigen können,
- für alle Schnittstellenelemente und den Attributen von Parametern einer Operation Dokumentationen erfassen können,
- Dokumentationen für nicht sichtbare Elemente erfassen können,
- adressatenspezifische Dokumentation unterstützen,
- die Erfassung von thematischen Rollen und Sichtbarkeitsbereichen ermöglichen,
- sich in eine bestehende Entwicklungsumgebung integrieren,
- die Dokumentation in den Quelltext schreiben und es sollte
- erweiterbar um weitere unterstützte Programmier- und Beschreibungs-sprachen sein.

Wie unter anderem in dem Fallbeispiel gesehen werden kann, erfüllt iDocIt! diese Anforderungen. Abbildung 7.5 zeigt im UI von iDocIt! einige realisierte Anforderungen. In der grafischen Oberfläche, die sich in Eclipse integriert, können die Schnittstellenelemente, die Attribute von Parametertypen und nicht sichtbare Elemente (z. B. eine Berechnungsvorschrift, oder qualitative Eigenschaften) mit beliebig vielen Dokumentationsblöcken dokumentiert werden. Ein Dokumentations-

block besteht dabei aus einer thematischen Rolle, einem Sichtbarkeitsbereich und einer Menge von adressatenspezifischen Beschreibungen. Wenn ein Element einer Operation selektiert wird, wird das Verb vom Operationsbezeichner extrahiert und die dazugehörigen thematischen Raster angezeigt.

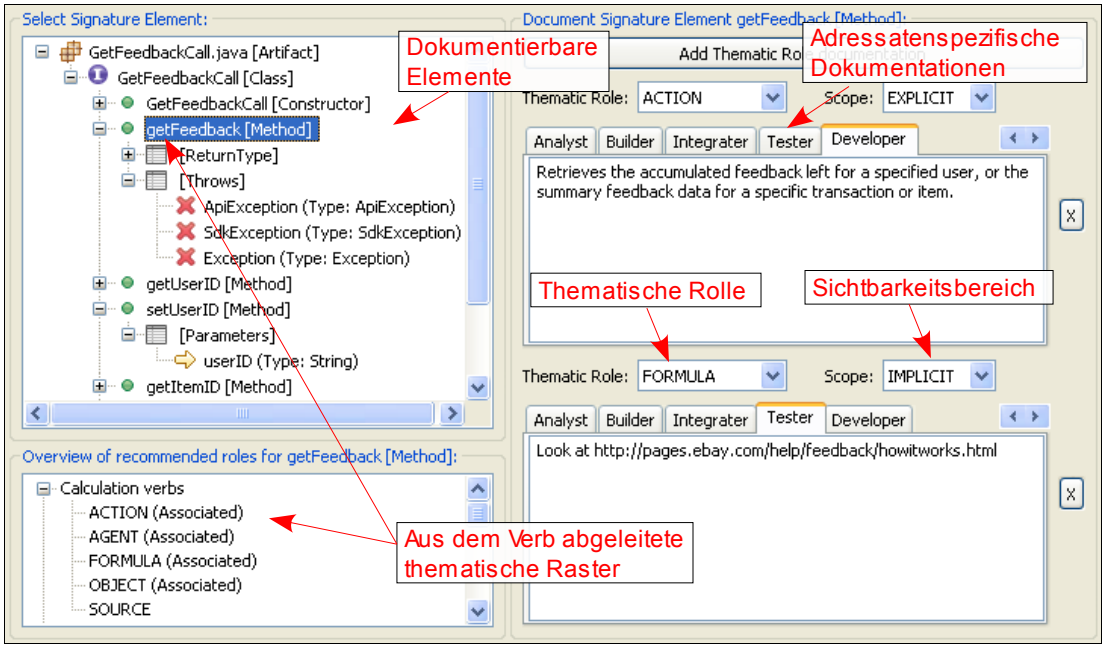


Abbildung 7.5: Realisierte Anforderungen im UI

Die im UI erfassten Dokumentationen werden dann direkt in die geöffnete Datei gespeichert. Die Erweiterbarkeit um unterstützte Programmier- und Beschreibungssprachen ist mit dem Eclipse Konzept der Extension Points und Extensions umgesetzt.

Die Anforderungen der AKRA GmbH (siehe Kapitel 3.4) werden auch erfüllt. Sie fordert, dass iDocIt! sich in Eclipse integrieren soll und dass es Java-Quellen, sowie WSDL Dateien dokumentieren kann.

7.3 Probleme des Prototypen

Wie in den Implementierungskapiteln beschrieben, sind die Parser-Erweiterungen für Java und WSDL noch nicht hundertprozentig ausgereift. Zum einen haben beide Parser das Problem mit Änderungen der Datei, die auftreten, wenn iDocIt! gerade

geöffnet ist. Die momentane Lösung, dass einfach die Änderungen überschrieben werden, sollte verbessert werden.

Das Formatierungsproblem betrifft auch beide Parser-Erweiterungen. Eclipse bietet zwar einen automatischen Formatierer an, den iDocIt! verwenden könnte, aber wenn ein Entwickler diesen nicht benutzt, würden seine ganzen händischen Formatierungen geändert werden. Dafür sollte noch eine bessere Lösung gefunden werden.

Bereits vorhandene Dokumentationen werden vom Prototypen bisher ignoriert und überschrieben. Wenn das Werkzeug in der Praxis eingesetzt werden sollte, ist aber davon auszugehen, dass bereits Dokumentation vorhanden ist. Deswegen muss ein Weg gefunden werden, die vorhandene Dokumentation durch eine Standardzuordnung in die Struktur von iDocIt! zu überführen.

Das größte Problem stellt aber WSDL4J dar, indem es nur eine feste Menge von Attributen unterstützt. Dieser Mangel muss auf jeden Fall behoben werden, bevor iDocIt! in der Praxis eingesetzt wird.

7.4 Fazit

In dem Fallbeispiel wurde sehr schön sichtbar wie iDocIt! dem Autor mit den thematischen Rastern einen Leitfaden an die Hand gibt, der zeigt, was alles dokumentiert werden sollte. Dieser Leitfaden steht immer zur Verfügung, unabhängig wie umfangreich die Methodensignatur ist (Anzahl der sichtbaren Elemente). Daran kann sich der Autor entlanghangeln und Stück für Stück die Schnittstelle vollständig dokumentieren. Durch die thematischen Rollen wird ihm aufgezeigt, worauf er seine Aufmerksamkeit lenken muss. Er wird dadurch besser unterstützt dokumentationswürdige Elemente zu identifizieren. Somit wird potentielle Stille in der Dokumentation vermieden.

Im Gegensatz dazu ist der Autor bei dem Raster von Javadoc auf sich alleine gestellt. Er muss die Schnittstelle durcharbeiten und (nicht sichtbare) Elemente selbst identifizieren, die dokumentiert werden sollten.

Dadurch, dass iDocIt! nicht die sichtbaren Schnittstellenelemente als Basis für seine Vorgaben nimmt, ist die Qualität der thematischen Raster unabhängig von der Implementierungsweise eines Entwicklers. Wie es z. B. bei der Methode `GetFeedback()` aus dem Fallbeispiel der Fall ist, werden die Einstellungen über andere Methoden getätigt und im Objekt gespeichert. Die Methode erhält keine direkten Eingaben. Im Gegensatz dazu könnten die gesamten Einstellungen als Parameter in die Methode hineingehen. Dann würde Javadoc auch ein besseres Raster vorgeben.

Außerdem geht die Qualität der thematischen Raster über die eines Rasters von Javadoc hinaus. Weil das Verb einem bestimmten Kontext zugeordnet wird, sind bereits Gedanken in seine Interpretation geflossen, es werden nicht nur die Rohdaten, also z. B. die einzelnen Namen der Elemente einer Methodensignatur, verwendet. Bei einem thematischen Raster hat der Autor also bereits Informationen über eine Methode erhalten und muss nicht nur mit den einfachen Daten arbeiten.

Damit die thematischen Raster gut funktionieren, muss daraufhin eine Schnittstelle entwickelt werden. Jede Operation sollte eine starke Kohäsion haben. D. h. sie sollte genau eine wohldefinierte Aufgabe haben. Für eine wohldefinierte Aufgabe kann dann ein präzises Verb verwendet werden. Wenn immer dasselbe Verb z. B. *to get* verwendet wird oder eine Operation mehrere Funktionen abdeckt, kann der Vorteil der thematischen Raster wieder zunichte gemacht werden.

Entwickler, aber auch Autoren, müssen dahingehend geschult werden, dass sie den Dokumentationsansatz mit thematischen Rollen umsetzen können und verinnerlichen. Das Arbeiten nach dem Ansatz ist nicht intuitiv möglich, das Konzept muss dazu verstanden worden sein.

Durch das Zuordnen von thematischen Rollen kann ein Element auch bereits ausreichend dokumentiert sein, wodurch ein beschreibender Text entfällt (siehe Kapitel 2). Dadurch kann eine Dokumentation gekürzt werden und auch Rauschen in der Dokumentation vermieden werden.

Die Navigation zu den einzelnen Schnittstellenelementen in dem iDocIt! Editor ist in dem aktuellen Prototypen viel „Klickarbeit“. Jede Methode, jeder Parameter und auch deren Attribute müssen einzeln in dem Navigationsbaum angeklickt werden, um dann Dokumentationsblöcke hinzufügen zu können.

Auch die gespeicherte Dokumentation ist auf den ersten Blick umfangreicher. Das liegt aber hauptsächlich daran, dass auch mehr dokumentiert wird als vorher. Wenn eine sehr umfassende API Dokumentation, wie z. B. die von der eBay Trading API, auf eine iDocIt! Dokumentation umgestellt wird, sollte sie potentiell kürzer ausfallen.

Das Lokalitätsprinzip für Dokumentation wird von iDocIt! auch umgesetzt. Dokumentationen werden dort erfasst, wo sie benötigt werden. Das wird z. B. bei den Beschreibungen der Attribute von Parametertypen deutlich. Ein Datentyp kann in verschiedenen Kontexten eingesetzt werden und hat jeweils eine andere Bedeutung. Jeder Kontext verlangt eine eigene Beschreibung, die am Verwendungsort stehen sollte und nicht mit Fallunterscheidungen direkt beim Datentyp.

Die konzeptionelle Unterstützung eines Autors beim Dokumentieren einer Schnittstelle ist also auf jeden Fall spürbar. Das Benutzungsmodell sollte aber noch verfeinert und auch erweitert werden.

8 Abschlussbetrachtungen

Abschließend zu dieser Bachelorarbeit wird noch einmal das Ergebnis zusammengefasst und ein Ausblick für dieses Thema und das Werkzeug iDocIt! gewagt. Weiterhin werden noch mögliche Weiterentwicklungen für iDocIt! vorgestellt.

8.1 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde evaluiert, inwieweit existierende Dokumentationssysteme und hauptsächlich Dokumentationswerkzeuge den Ansatz unterstützen, Dokumentationen mit thematischen Rastern und Rollen zu erstellen. Dazu wurden 19 Dokumentationssysteme im Internet recherchiert und deren Funktionen und Eigenschaften anhand eines umfassenden Kriterienkatalogs überprüft.

Es wurde festgestellt, dass die meisten Systeme die allgemeinen Kriterien erfüllen, aber nicht die speziellen Kriterien, zur Dokumentation thematischer Rollen und adressatenspezifischen Beschreibungen.

Da kein Dokumentationssystem alle Anforderungen erfüllt, wurde das Werkzeug iDocIt! als Eclipse Plugin entwickelt. Für den Prototypen wurden Parser-Erweiterungen für Java und WSDL entwickelt, sodass diese dokumentiert werden können.

In einem Fallbeispiel hat iDocIt! dann seine Vorzüge gezeigt. Ein Autor erhält durch die thematischen Raster in iDocIt! einen guten Leitfaden als Unterstützung beim Dokumentieren. Dadurch und auch weil teilweise zur Dokumentation nur thematische Rollen zugeordnet werden brauchen, kann eine Schnittstelle schneller dokumentiert werden. Weiterhin werden potentielle Stille und auch Rauschen durch die Vorgaben der Raster bestmöglich vermieden.

Die thematischen Raster sind auch gleichzeitig eine Vollständigkeitsanzeige für eine O-Doku. Bereits zugeordnete Rollen werden in iDocIt! gekennzeichnet, sodass der Autor immer einen Überblick über die dokumentierten Elemente hat.

Auch kann eine Dokumentation durch die Verwendung von thematischen Rollen potentiell gekürzt werden. Eine Rolle kann u. U. das aussagen, was sonst in einigen Sätzen beschrieben werden würde. Durch die kürzeren und prägnanteren Dokumentationen soll in der Projektarbeit die Einarbeitung von Mitarbeitern erleichtert werden.

Die grafische Oberfläche von iDocIt! ermöglicht Autoren, die nicht in Quelltexten direkt arbeiten möchten, Dokumentationen zu erstellen.

Diese Bachelorarbeit hat einen Prototypen als Ergebnis, für den es noch viele Weiterentwicklungsmöglichkeiten und Potential gibt.

Die AKRA GmbH kann ihre Mitarbeiter in diesem Ansatz und dem Werkzeug schulen, damit diese einen Vorteil im Projektgeschäft geltend machen können. IDocIt! eröffnet für die AKRA GmbH aber auch einen völlig neuen Geschäftszweig: „Documentation-Consulting“. Sie könnte Dienstleistungen für die Erstellung von APIs und deren Dokumentationen anbieten. Auch Audits bestehender API-Dokumentation wären eine neue denkbare Dienstleistung.

Wie in der Motivation der Bachelorarbeit erwähnt, wird iDocIt! als Open Source-Projekt publiziert. Der Ansatz und iDocIt! sollen daher in Artikeln von bekannten Zeitschriften bekannt gemacht werden und die Software weiter verbessert werden.

U. a. interessierte Unternehmen können dann Parser-Erweiterungen für ihre Zwecke entwickeln und auf der Projektseite von iDocIt! (http://code.google.com/p/idocit/) veröffentlichen. Die gewählte Modularisierung von iDocIt! soll dieses Vorgehen und die verteilte Weiterentwicklung vorantreiben.

8.2 Mögliche Weiterentwicklungen

In den Kapiteln zur Implementierung und Systembewertung wurden bereits einige nötige Weiterentwicklungen aufgezeigt. Das waren u. a., dass

- WSDL4J alle Attribute unterstützt,
- Formatierungen im Quelltext erhalten bleiben,
- das Verhalten von iDocIt! verbessert wird, wenn die Quelldatei während der Bearbeitung in iDocIt! geändert wurde (z. B. dynamisches Aktualisieren der Oberfläche mit automatischer Zuordnung der Änderungen) und dass
- vorhandene Dokumentationen von iDocIt! behandelt und in sein Format überführt werden.

Aber es gibt auch noch viele weitere mögliche Weiterentwicklungen, die hauptsächlich in die Richtung der besseren Benutzbarkeit von iDocIt! gehen:

- **Vollständigkeitsanalyse von Dokumentationen:** In der Oberfläche soll die Vollständigkeit der Dokumentation visualisiert werden. Das kann mit einer Prozentzahl geschehen, die sich aus den empfohlenen thematischen Rollen und den davon bereits dokumentierten Rollen errechnet. Zusätzlich könnten die Symbole in dem Navigationsbaum für die Schnittstellenelemente verschiedenfarbig gestaltet werden, je nach Vollständigkeitsgrad der Dokumentation. Es könnte z. B. ein rotes Symbol verwendet werden, wenn eine Operation noch nicht dokumentiert ist, ein gelbes Symbol, wenn mindestens eine Rolle, aber noch nicht alle dokumentiert sind und ein grünes Symbol, wenn alle empfohlenen thematischen Rollen zugeordnet worden sind.
- **Exportfunktion:** Die Dokumentation soll aus der grafischen Oberfläche heraus in ein anderes Format exportiert werden. Das Format könnte z. B. DITA oder DocBook sein, wofür wiederum viele Exportvarianten existieren.

Über dieses (Zwischen-)Format kann die Dokumentation in viele andere Formate exportiert werden, wie z. B. HTML und PDF.

- **Schnelleres Dokumentieren mit iDocIt!:** Hierbei soll das Benutzungsmodell erweitert werden, damit ein Autor schneller eine Dokumentation erstellen kann.
  - **Dokumentieren im Quelltexteditor:** Bei dieser Variante kann der Autor z. B. in dem Java-Editor in Eclipse über ein Tastenkürzel oder dem Kontextmenü ein kleines Fenster öffnen, das die Dokumentationsfelder nur für eine Methode enthält. Dadurch kann die Dokumentation für eine Methode schnell erfasst werden, was i. d. R. von Entwicklern als positiv angesehen wird. Falls doch mehr dokumentiert werden soll, müsste der vollständige iDocIt!-Editor geöffnet werden.
  - **Sichtbare Elemente einer Methode zusammen dokumentieren:** In der Oberfläche sollen die sichtbaren Elemente einer Methodensignatur in einem Fenster angezeigt werden, wenn eine Methode selektiert wurde. Die sichtbaren Elemente sich die am häufigsten dokumentierten Elemente, die dann in einem Fenster dokumentiert werden. Dadurch wird dem Autor das Navigieren durch den Navigationsbaum der Schnittstellenelemente erspart.
- **Rollenabhängige Editoren:** Für die Erfassung der Dokumentationen in der grafischen Oberfläche sollen nicht nur Freitextfelder verwendet werden. Abhängig von der zugeordneten thematischen Rolle, soll ein entsprechender Editor erscheinen. Beispielsweise könnte zu der Rolle FORMULA ein Formeleditor zur Eingabe einer Formel angezeigt werden.

Weiterhin könnte analysiert werden wie Entwickler und Autoren arbeiten und danach könnte das Benutzungsmodell weiter angepasst werden.

A Auswahl existierender Dokumentationssysteme

Name	Software Lizenz	Kategorie	Speicherort der Dokumentation	Format		Unterstützt Javadoc Format
				XML	Schlüssel-wörter	
Help Generator	Proprietär	1	externe Datei	Nein	Nein	Nein
Imagix 4D	Proprietär	1	externe Datei	Nein	Nein	Nein
Universal Report	Proprietär	1	externe Datei	Nein	Nein	Nein
RDoc	Ruby License	2	im Quelltext	Nein	Ja	Nein
CppDoc	Open Source	3	im Quelltext	Nein	Ja	Ja
Doc-O-Matic	Proprietär	3	im Quelltext	Ja	Ja	Ja
DOC++	GPL	3	im Quelltext	Nein	Ja	Nein
Document! X	Proprietär	3	im Quelltext	Ja	Nein	Nein
Doxygen	GPL	3	im Quelltext	Ja	Ja	Ja
GenHelp	Proprietär	3	im Quelltext	Nein	Ja	Ja
HeaderDoc	APSL	3	im Quelltext	Nein	Ja	Nein
Javadoc	GPL	3	im Quelltext	Nein	Ja	Ja
Natural Docs	GPL	3	im Quelltext	Nein	Ja	Ja
ROBODoc	GPL	3	im Quelltext	Nein	Ja	Nein
TwinText	Proprietär	3	im Quelltext	Nein	Ja	Nein
VSdocman	Proprietär	3	im Quelltext	Ja	Nein	Nein
YARD	MIT License	3	im Quelltext	Nein	Ja	Nein
DITA	Open Source	4	externe Datei	Ja	Nein	Nein
DocBook	GNU FDL	4	externe Datei	Ja	Nein	Nein

Tabelle A.1: Kategorien und allgemeine Informationen zu den untersuchten Dokumentationssystemen

Name	Unterstützte Dokumentationstypisierungen						
	Eingabe	**Ausgabe**	**Fehler**	**Autor**	**Version**	**Seit welcher Version enthalten**	**Veraltet**
Help Generator	-	-	-	-	-	-	-
Imagix 4D	-	-	-	-	-	-	-
Universal Report	-	-	-	-	-	-	-
RDoc	-	-	-	-	-	-	-
CppDoc	+	+	+	+	+	+	+
Doc-O-Matic	+	+	+	+	+	+	+
DOC++	+	+	+	+	+	+	+
Document! X	+	+	+	o	o	o	o
Doxygen	+	+	+	+	+	+	+
GenHelp	+	+	+	+	+	+	+
HeaderDoc	+	+	+	+	+	+	+
Javadoc	+	+	+	+	+	+	+
Natural Docs	+	+	+	+	+	o	+
ROBODoc	+	+	+	+	o	o	o
TwinText	+	+	+	+	+	-	+
VSdocman	+	+	+	+	+	+	o
YARD	+	+	+	+	+	+	+
DITA	o	o	o	o	o	o	o
DocBook	o	o	o	o	o	o	o

Tabelle A.2: Unterstützte Dokumentationstypisierungen von den untersuchten Dokumentationssystemen

Legende:

+ ist vorgesehen
- nicht möglich
o nicht vorgesehen, aber möglich

Name	Unterstützte Dokumentationstypisierungen				
	Referenz	**Beschreibung**	**Untypisierter Bereich (Freitext)**	**Zielgruppen-gerichtete Beschreibung**	**Erweiterbar um Typen**
Help Generator	-	-	-	-	-
Imagix 4D	-	-	-	-	-
Universal Report	-	-	-	-	-
RDoc	-	-	+	-	-
CppDoc	+	-	+	-	-
Doc-O-Matic	+	+	+	o	+
DOC++	+	+	+	-	-
Document! X	+	+	-	o	+
Doxygen	+	+	+	o	+
GenHelp	+	+	+	o	+
HeaderDoc	+	+	+	o	+
Javadoc	+	o	+	o	+
Natural Docs	+	+	+	o	+
ROBODoc	+	+	-	o	+
TwinText	+	+	-	-	-
VSdocman	+	+	+	o	+
YARD	+	+	+	o	+
DITA	o	o	o	o	o
DocBook	o	o	o	o	o

Tabelle A.3: Fortsetzung: Unterstützte Dokumentationstypisierungen von den untersuchten Dokumentationssystemen

Name	Nicht sichtbare Elemente	
	abbildbar	**ableitbar**
Help Generator	-	-
Imagix 4D	-	-
Universal Report	-	-
RDoc	-	-
CppDoc	o	-
Doc-O-Matic	o	-
DOC++	-	-
Document! X	o	-
Doxygen	o	-
GenHelp	o	-
HeaderDoc	+	-
Javadoc	o	-
Natural Docs	o	-
ROBODoc	+	-
TwinText	-	-
VSdocman	o	-
YARD	o	-
DITA	o	-
DocBook	o	-

Tabelle A.4: Unterstützung nicht sichtbarer Elemente von den untersuchten Dokumentationssystemen

Name	Unterstützte Programmier- und Beschreibungssprachen			
	Java	**WSDL**	**Anzahl insgesamt**	**Erweiterbar für Sprachen mit Kommentaren**
CppDoc	Nein	Nein	1	Nein
Doc-O-Matic	Ja	Nein	11	Ja
DOC++	Ja	Nein	3	Nein
Document! X	Nein	Ja	4	Nein
Doxygen	Ja	Nein	11	Nein
GenHelp	Nein	Nein	3	Nein
HeaderDoc	Ja	Nein	9	Nein
Help Generator	Nein	Nein	4	Nein
Imagix 4D	Ja	Nein	2	Nein
Javadoc	Ja	Nein	1	Nein
Natural Docs	Teilweise	Nein	19	Ja
RDoc	Nein	Nein	1	Nein
ROBODoc	Ja	Nein	15	Ja
TwinText	Ja	Nein	25	Ja
Universal Report	Ja	Nein	25	Ja
VSdocman	Nein	Nein	2	Nein
YARD	Nein	Nein	1	Nein

Tabelle A.5: Unterstützte Programmier- und Beschreibungssprachen von den untersuchten Dokumentationssystemen

Name	URL
CppDoc	http://www.cppdoc.com
DITA	http://dita.xml.org
Doc-O-Matic	http://www.doc-o-matic.com
DOC++	http://docpp.sourceforge.net
DocBook	http://www.docbook.org
Document! X	http://www.innovasys.com/products/dx2010/overview.aspx
Doxygen	http://www.doxygen.org
GenHelp	http://www.frasersoft.net/genhelp
HeaderDoc	http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/HeaderDoc/intro/intro.html
Help Generator	http://www.helpgenerator.com
Imagix 4D	http://www.imagix.com
Javadoc	http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
Natural Docs	http://www.naturaldocs.org
RDoc	http://rdoc.sourceforge.net
ROBODoc	http://www.xs4all.nl/~rfsber/Robo/robodoc.html
TwinText	http://www.ptlogica.com/TwinText/
Universal Report	http://www.omegacomputer.com
VSdocman	http://www.helixoft.com/vsdocman/overview.html
YARD	http://yardoc.org

Tabelle A.6: Links zu den untersuchten Dokumentationssystemen

B WSDL 1.1 Spezifikation

Die in Tabelle B.1 und Tabelle B.2 verwendeten Datentypen `qname`, `nmtoken` und `uri` gehören zu den eingebauten Datentypen `QName`, `NMTOKEN` und `anyURI` der XML Schema Definition.[73]

Legende:

? Optional (0 bis 1 Mal vorhanden)
* Mehrfach vorhanden (0 bis beliebig oft)

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

  <import namespace="uri" location="uri"/> *

  <wsdl:documentation .... /> ?

  <wsdl:types> ?
    <wsdl:documentation .... /> ?
    <xsd:schema .... />*
    <-- extensibility element --> *
  </wsdl:types>

  <wsdl:portType name="nmtoken"> *
    <wsdl:documentation .... /> ?
    <wsdl:operation name="nmtoken"> *
      <wsdl:documentation .... /> ?
      <wsdl:input name="nmtoken"? message="qname"> ?
        <wsdl:documentation .... /> ?
      </wsdl:input>
      <wsdl:output name="nmtoken"? message="qname"> ?
        <wsdl:documentation .... /> ?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname"> *
        <wsdl:documentation .... /> ?
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>
```

Tabelle B.1: WSDL 1.1 Spezifikation[74]

73 Vgl. Biron, Paul V / Malhotra, Ashok: XML Schema Part 2, 2001
74 Vgl. Christensen, Erik et al.: WSDL 1.1, 2001

```
  <wsdl:message name="nmtoken"> *
    <wsdl:documentation .... /> ?
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </wsdl:message>

  <wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation .... />?
    <-- extensibility element --> *
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation .... /> ?
      <-- extensibility element --> *
      <wsdl:input> ?
        <wsdl:documentation .... /> ?
        <-- extensibility element -->
      </wsdl:input>
      <wsdl:output> ?
        <wsdl:documentation .... /> ?
        <-- extensibility element --> *
      </wsdl:output>
      <wsdl:fault name="nmtoken"> *
        <wsdl:documentation .... /> ?
        <-- extensibility element --> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="nmtoken"> *
    <wsdl:documentation .... />?
    <wsdl:port name="nmtoken" binding="qname"> *
      <wsdl:documentation .... /> ?
      <-- extensibility element -->
    </wsdl:port>
    <-- extensibility element -->
  </wsdl:service>

  <-- extensibility element --> *

</wsdl:definitions>
```

Tabelle B.2: Fortsetzung: WSDL 1.1 Spezifikation

C Java Interface Deklaration

In Tabelle C.1 ist die Spezifikation einer Java Interface Deklaration (aus der Java Sprachspezifikation der 3. Edition) auszugsweise in der EBNF aufgeführt. Es sind nicht alle Elemente der Spezifikation in dem Auszug enthalten und die Produktionen werden teils nicht vollständig bis zum Terminalsymbol aufgeführt. Die groß geschriebenen Wörter (z. B. `InterfaceDeclaration` und `Identifier`) stellen Nichtterminalsymbole und die klein geschriebenen Wörter (z. B. `long` und `public`), sowie die Zeichen in einfachen Hochkommata (z. B. `'{'` und `';'`), stellen Terminalsymbole dar. Ein vertikaler Strich „|“ trennt Alternativen, mit eckigen Klammern „`[...]`“ eingeschlossene Symbole sind optional (ein- oder keinmal vorhanden) und mit geschweiften Klammern eingeschlossene Symbole „`{...}`“ können null bis beliebig oft wiederholt werden.

Legende:

'...'	: Terminalsymbol (Zeichen in einfachen Hochkommata)
symbolName	: Terminalsymbol (klein geschriebene Wörter)
SymbolName	: Nichtterminalsymbole (groß geschriebene Wörter)
[...]	: Optionale Symbole (ein- oder keinmal vorhanden)
{ ... }	: Optionale Wiederholung von Symbolen (null bis beliebig oft)
... | ...	: Alternative Symbole

```
InterfaceDeclaration = NormalInterfaceDeclaration

NormalInterfaceDeclaration = { InterfaceModifier } interface
                              Identifier [ ExtendsInterfaces ]
                              InterfaceBody

InterfaceModifier = public | protected | private | abstract
                    | static

ExtendsInterfaces = extends InterfaceType { ',' InterfaceType }

InterfaceBody = '{' { InterfaceMemberDeclaration } '}'

InterfaceMemberDeclaration = InterfaceDeclaration
                              | AbstractMethodDeclaration
                              | ClassDeclaration | ';'

InterfaceType = TypeDeclSpecifier

ClassType = TypeDeclSpecifier

TypeDeclSpecifier = TypeName | ClassOrInterfaceType '.' Identifier

TypeName = Identifier | TypeName '.' Identifier

ClassOrInterfaceType = ClassType | InterfaceType

AbstractMethodDeclaration = { AbstractMethodModifier } ResultType
                             MethodDeclarator [ Throws ] ';'

AbstractMethodModifier = public | abstract

ResultType = Type | void

Throws = throws ExceptionType { ',' ExceptionType }

ExceptionType = ClassType | TypeVariable

TypeVariable = Identifier

MethodDeclarator = Identifier '(' [ FormalParameterList ] ')'

FormalParameterList = { FormalParameters ',' } LastFormalParameter

LastFormalParameter = [ final ] Type [ ... ] VariableDeclaratorId
                      | FormalParameter

FormalParameter = [ final ] Type VariableDeclaratorId

VariableDeclaratorId = Identifier | VariableDeclaratorId '[' ']'

Type = PrimitiveType | ReferenceType

PrimitiveType = byte | short | int | long | char | float
                | double | boolean

ReferenceType = ClassOrInterfaceType | TypeVariable | ArrayType

ArrayType = Type '[' ']'

ClassDeclaration = NormalClassDeclaration | EnumDeclaration

NormalClassDeclaration = { ClassModifier } class Identifier
                         [ Super ] [ Interfaces ] ClassBody
```

```
ClassModifier = public | protected | private | abstract
                | static | final

Super = extends ClassType

Interfaces = implements InterfaceType { ',' InterfaceType }

ClassBody = '{' { ClassBodyDeclaration } '}'

ClassBodyDeclaration = ClassMemberDeclaration
                       | ConstructorDeclaration

ClassMemberDeclaration = MethodDeclaration | FieldDeclaration
                         | ClassDeclaration | InterfaceDeclaration
                         | ';'

MethodDeclaration = MethodHeader | MethodBody

MethodHeader = { MethodModifier } ResultType MethodDeclarator
               [ Throws ]

MethodModifier = public | protected | private | abstract
                 | static | final | synchronized

FieldDeclaration = { FieldModifier } Type VariableDeclarators ';'

VariableDeclarators = VariableDeclarator { ',' VariableDeclarator }

VariableDeclarator = VariableDeclaratorId
                     [ '='  VariableInitializer ]

VariableDeclaratorId = Identifier | VariableDeclaratorId [ ]

ConstructorDeclaration = { ConstructorModifier }
                         ConstructorDeclarator [ Throws ]
                         ConstructorBody

ConstructorModifier = public | protected | private

ConstructorDeclarator = SimpleTypeName '('
                        [ FormalParameterList ] ')'
```

Tabelle C.1: Auszug der Java Sprachspezifikation (3. Edition) für eine Interface Deklaration[75] in EBNF

75 Vgl. Gosling, James et al.: Java Language Specification, 2005, S. 260ff

D Benutzerhandbuch

D.1 Systemvoraussetzungen

Das Dokumentationswerkzeug iDocIt! ist ein in Java entwickeltes Plugin für Eclipse. Es benötigt eine installierte Version von

- Eclipse 3.6 und
- Java Runtime Environment (JRE) oder Java Development Kit (JDK) in der Version 1.6 (bzw. 6).

Für das Ableiten der thematischen Raster aus einem Verb werden

- das Wörterbuch von WordNet in der Version 3.0 und
- ein Model des Stanford Part-Of-Speech Taggers in der Version 3.0

benötigt.

WordNet 3.0 kann von URL http://wordnet.princeton.edu/wordnet/download/ heruntergeladen werden. Dort wählen Sie am besten „Download tar-gzipped: WordNet-3.0.tar.gz“.

Den Stanford Part-Of-Speech Tagger in der Version 3.0 laden Sie bitte von der URL http://nlp.stanford.edu/software/tagger.shtml herunter. Dort wählen Sie „Download basic English Stanford Tagger version 3.0 (re-entrant version) [17 MB] (highly recommended!)“. Die Basic Version ist ausreichend, da diese das benötigte Model „left3words-wsj-0-18.tagger“ enthält.

Alle weiteren Voraussetzungen leiten sich aus denen von Eclipse und Java ab, die i. d. R. auf allen handelsüblichen PCs lauffähig sind.

D.2 Installation

Zur Installation von den iDocIt! Plugins in Eclipse, werden die Dateien

- de.akra.idocit_X.X.X.jar
- de.akra.idocit.ui_X.X.X.jar
- de.akra.idocit.java_X.X.X.jar
- de.akra.idocit.wsdl_X.X.X.jar

in das „plugins“ Verzeichnis der Eclipse Installation kopiert (z. B. C:\Programme\eclipse\plugins\). Das „X.X.X“ in dem Dateinamen steht für die Versionsnummer der einzelnen Plugins.

Danach kann Eclipse gestartet werden und die Plugins werden automatisch erkannt und geladen.

D.3 Bedienungsanleitung

D.3.1 Konfiguration von iDocIt!

Bevor iDocIt! verwendet werden kann, müssen einige Einstellungen getätigt werden. Öffnen Sie dazu in Eclipse über „Window“ → „Preferences“ → „iDocIt!“ die Preference Page von iDocIt!. In dem ersten Fenster der Einstellungen muss der Pfad zum Wörterbuch von WordNet 3.0 gesetzt werden, sowie zu dem Model des Stanford Part-Of-Speech Taggers in der Version 3.0.

Wenn Sie die beiden Programme noch nicht heruntergeladen haben, machen Sie dieses und entpacken die heruntergeladenen Dateien (siehe Kapitel D.1 Systemvoraussetzungen).

Geben Sie den Pfad zum Verzeichnis „...\WordNet-3.0\dict\“ in das Feld „WordNet Path“ ein und in das Feld „PoS-Tagger Model“ den Pfad zu der Datei „left3words-wsj-0-18.tagger“ an. In Abbildung D.1 ist dieses exemplarisch gezeigt.

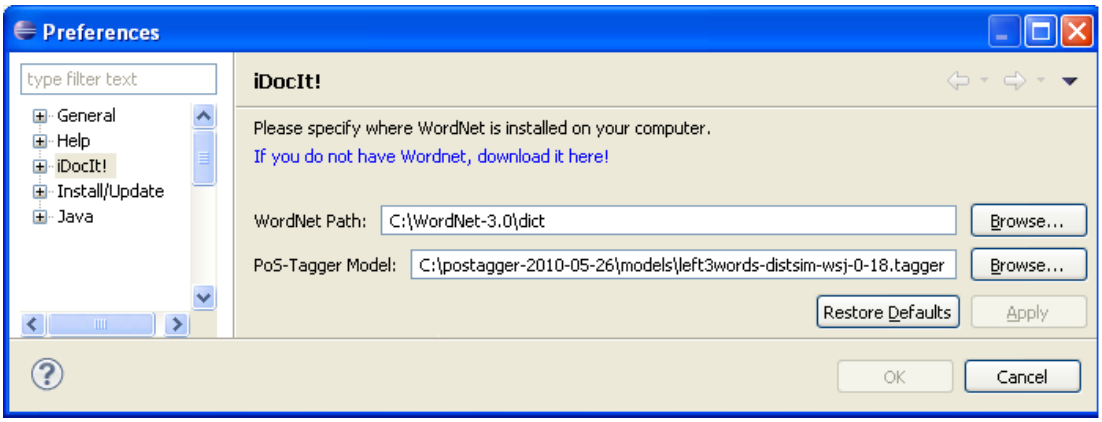


Abbildung D.1: iDocIt! Preference Page

In der Preference Page für die Adressaten (siehe Abbildung D.2) können die verwendeten Adressaten verwaltet werden. Für einen Adressaten kann ein Name und eine Beschreibung erfasst werden.

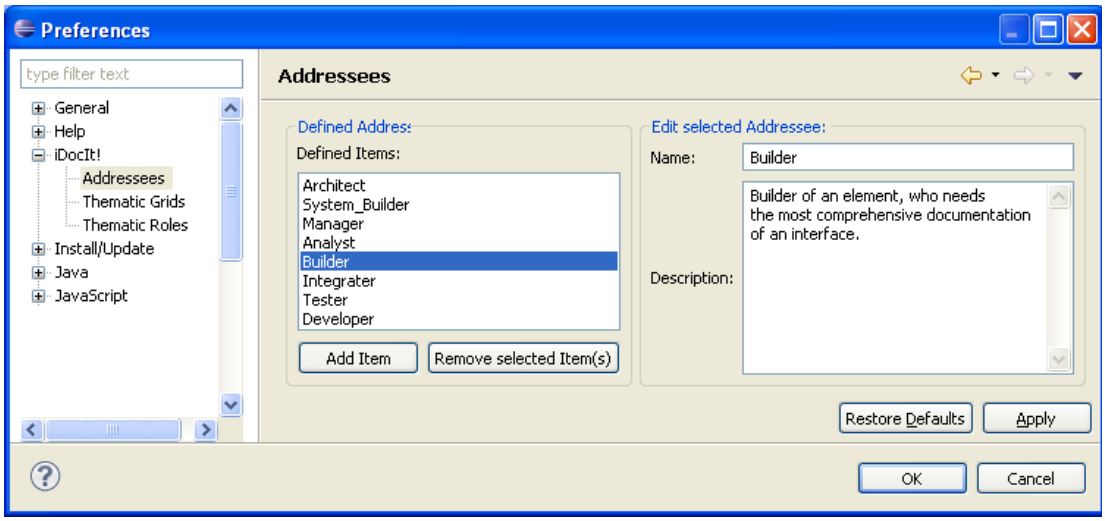


Abbildung D.2: iDocIt! Preference Page für Adressaten

In der Preference Page für thematische Raster (siehe Abbildung D.3) werden die thematischen Raster verwaltet. Hier können neue Raster erstellt oder vorhandene gelöscht werden.

Zu einem thematischen Raster kann ein Name erfasst werden, sowie eine Beschreibung, eine kommaseparierte Liste von Verben, die diesem Raster zugeordnet sind und eine Menge von thematischen Rollen, die in dem Raster enthalten sein sollen. Die thematischen Rollen, die diesem Raster zugeordnet sein sollen, werden mit einem Haken markiert. Diese werden dann in der iDocIt! Oberfläche als empfohlene thematische Rollen für die zugeordneten Verben angezeigt.

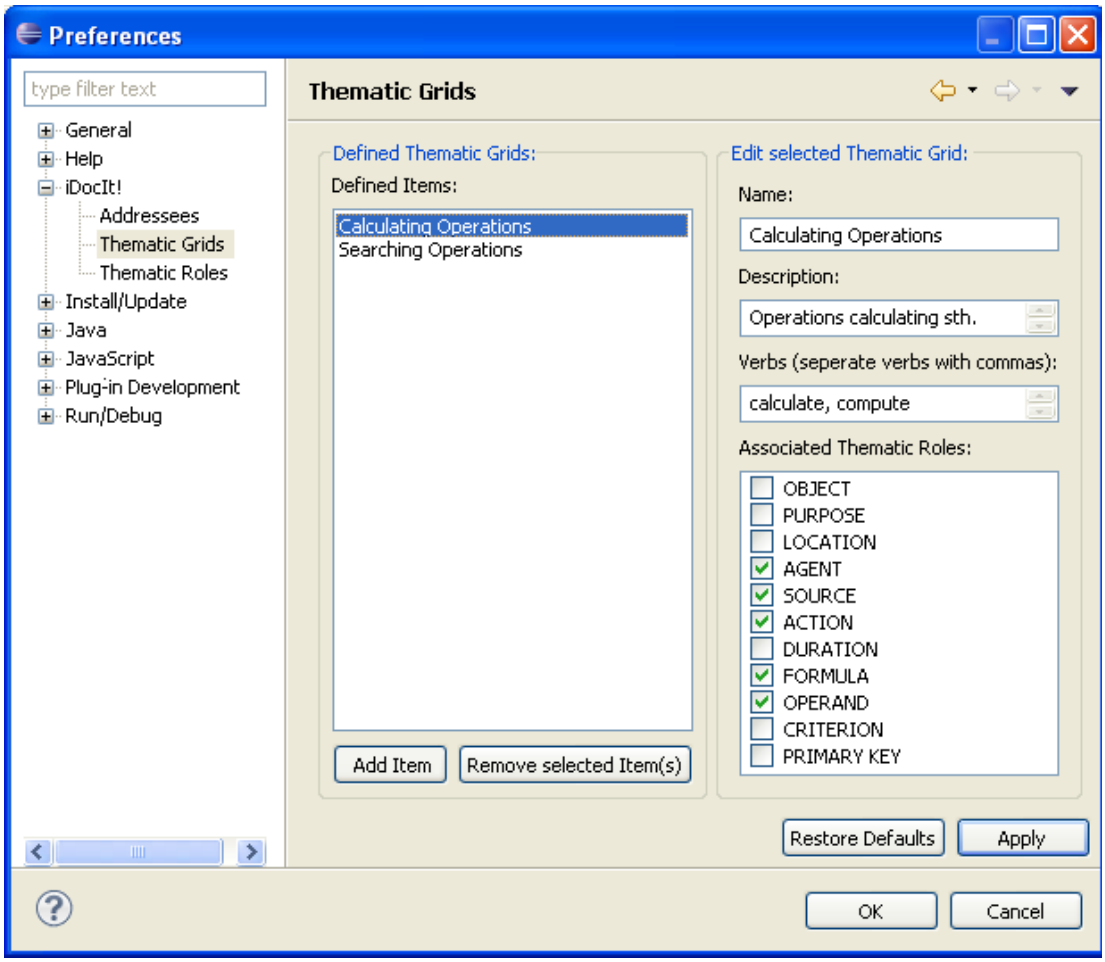


Abbildung D.3: iDocIt! Preference Page für thematische Raster

Die thematischen Rollen, die für die thematischen Raster ausgewählt werden können, werden in der Preference Page für thematische Rollen (siehe Abbildung D.4) verwaltet. Dort können sie hinzugefügt und gelöscht werden.

Zu einer thematischen Rolle kann ein Name und eine Beschreibung dieser Rolle erfasst werden. Die Namen der thematischen Rollen sollten einheitlich in Großbuchstaben geschrieben sein.

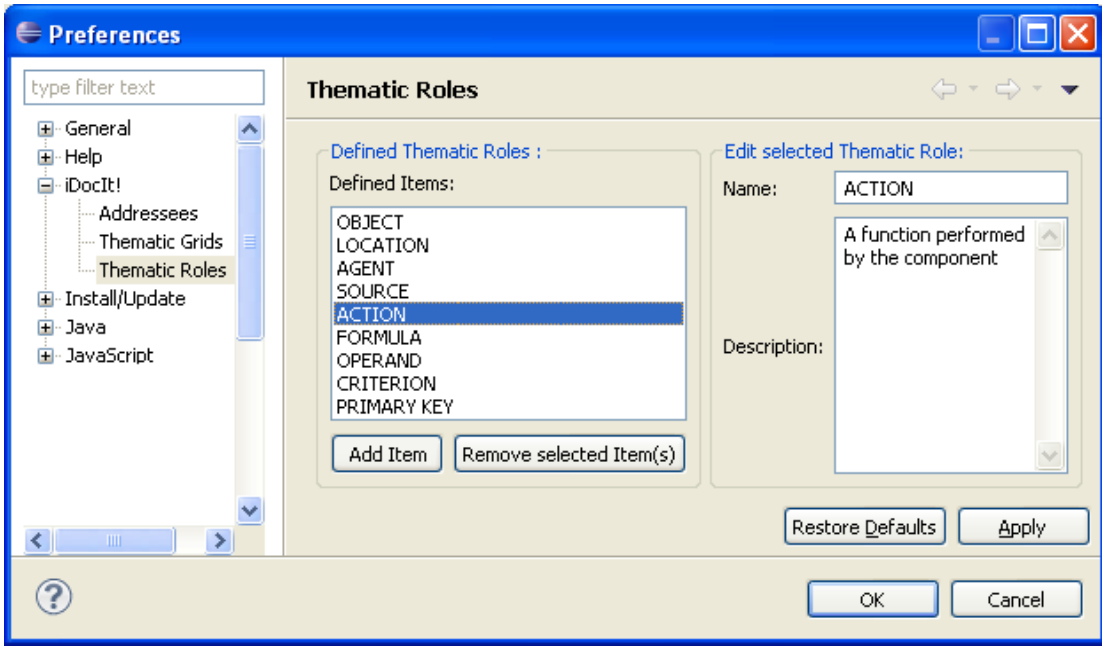


Abbildung D.4: iDocIt! Preference Page für thematische Rollen

D.3.2 Erstellen von Dokumentationen

Wenn eine Dokumentation für eine Schnittstelle erfasst werden soll, wird der iDocIt! Editor über das Kontextmenü der entsprechenden Datei in Eclipse geöffnet. In dem Kontextmenü klicken Sie dann auf „Open With“ und wählen dann „iDocIt!“. Wenn der Eintrag „iDocIt!“ nicht in der ersten Liste sein sollte, wählen Sie „Other...“ und dann aus der Liste „iDocIt!“. Falls der Eintrag auch dort nicht vorhanden sein sollte, ist das iDocIt! Plugin nicht korrekt installiert.

Schließen Sie dann Eclipse, kopieren die *.jar Dateien noch einmal in das „plugins“ Verzeichnis der Eclipse Installation und starten Eclipse erneut (siehe D.2 Installation).

Wenn der iDocIt! Editor ausgewählt werden kann, wird von iDocIt! geprüft ob der Dateityp unterstützt wird (Prüfung anhand der Dateiendung), d. h. ob eine Parser-Erweiterung für diesen Typ verfügbar ist. Sollte die Datei nicht unterstützt werden, wird dieses in der Oberfläche angezeigt (siehe Abbildung D.5).

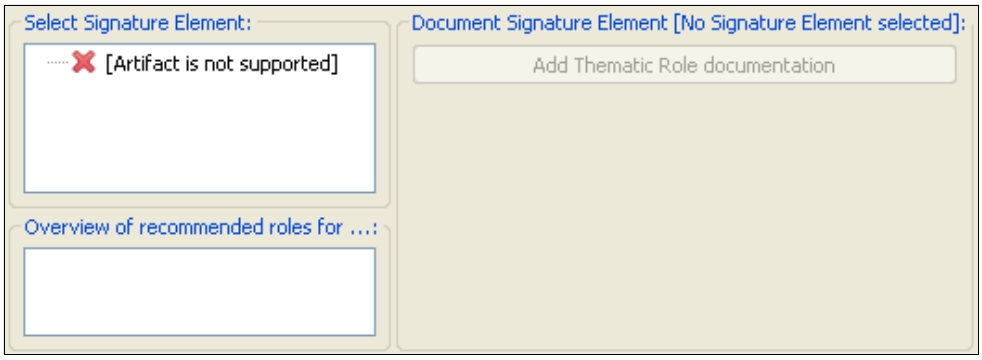

Select Signature Element:
[Artifact is not supported]
Overview of recommended roles for ...:
Document Signature Element [No Signature Element selected]:
Add Thematic Role documentation

Abbildung D.5: iDocIt! Editor mit nicht unterstützter Datei geöffnet

Falls eine Parser-Erweiterung für diesen Typ vorhanden ist, wird die Struktur der Schnittstelle angezeigt (siehe Abbildung D.6). Die Dokumentationen können dann für die einzelnen Schnittstellenelemente erfasst werden.

In dem Baum „Select Signature Element“ kann in der Schnittstellenstruktur navigiert werden. Klicken Sie das Element an, das Sie dokumentieren möchten.

Nach der Auswahl werden bereits gespeicherte Dokumentationen auf der rechten Seite in einer Liste von Dokumentationsblöcken gezeigt. Mit dem Klick auf die Schaltfläche „Add Thematic Role documentation“ kann ein Dokumentationsblock dem Signaturelement hinzufügt werden. Mit dem Klick auf die Schaltfläche „X“ wird der entsprechende Dokumentationsblock wieder aus der Liste entfernt.

In einem Dokumentationsblock kann eine „Thematic Role“ (thematische Rolle) und ein „Scope“ (Sichtbarkeitsbereich) ausgewählt werden, sowie für die hinterlegten Adressaten Beschreibungen estellt werden.

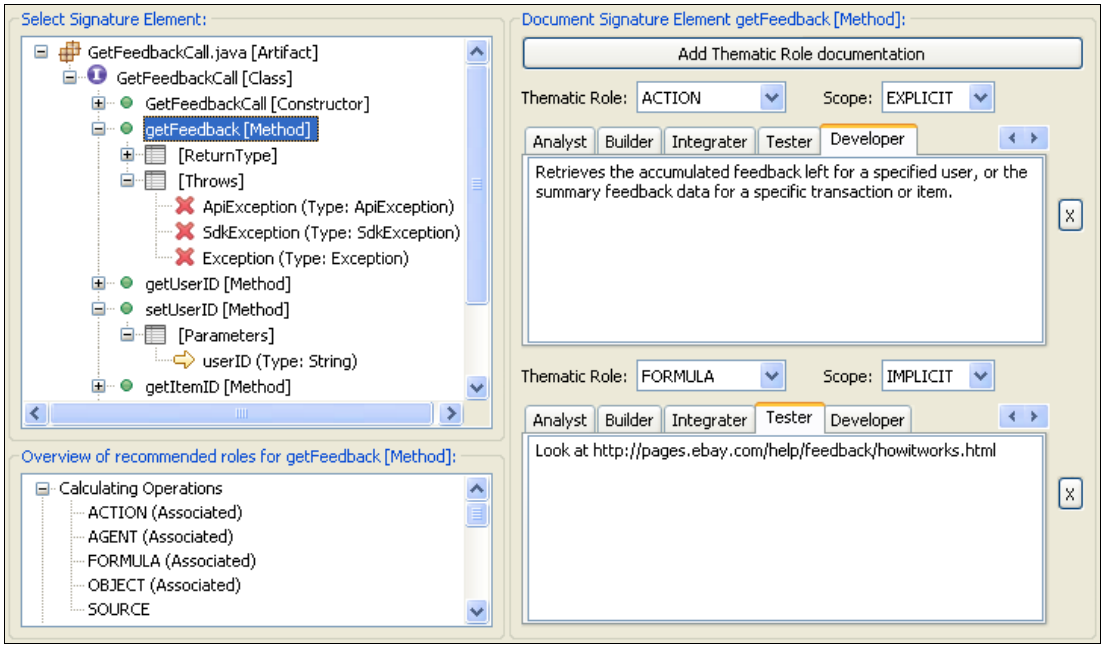


Abbildung D.6: iDocIt! Editor mit der geöffneten Klasse `GetFeedbackCall` *aus dem eBay SDK for Java*

Wenn ein Element einer Operation, d. h. die Operation selbst oder ein im Navigationsbaum hierarchisch darunter liegendes Element, selektiert wird, werden unten links im Fenster die thematischen Raster und Rollen zu dem Verb im Operationsbezeichner angezeigt. Ist ein anderes Element selektiert oder enthält die Operation kein Verb bzw. kein bekanntes englisches Verb, bleibt das Kontrollelement leer.

Werden die thematischen Raster angezeigt, werden auch die bereits zugeordneten thematischen Rollen als „Associated“ gekennzeichnet. Die vergebenen Rollen werden in den Dokumentationen von allen Elementen einer Operation, sowie in den Dokumentationen der Elemente, die auf dem direkten Weg von der Operation bis hoch zum Wurzelelement, dem „Artifact“ bzw. der Datei liegen. In dem Beispiel aus Abbildung D.6 werden die Dokumentationen aus den Elementen der Methode „getFeedback [Method]“ und auch die Dokumentationen von den Elementen „GetFeedbackCall [Class]“ und „GetFeedbackCall.java [Artifact]“ nach bereits zugeordneten Rollen überprüft.

Die Änderungen an den Dokumentationen werden mit der Speichern-Schalftfäche von Eclipse oder mit der Tastenkombination Strg+S gespeichert. Falls Änderungen in der Zwischenzeit an der Quelldatei getätigt wurden, werden sie mit dem Speichern, ohne einen Hinweis, überschrieben.

E Abkürzungsverzeichnis

API	Application Programming Interface
APSL	Apple Public Source License
ASB	Abstrakter Syntaxbaum
AST	Abstract Syntax Tree
DITA	Darwin Information Typing Architecture
DOM	Document Object Model
EBNF	Erweiterte Backus-Naur-Form
GNU	„**G**NU is **n**ot **U**nix“
GNU FDL	GNU Free Documentation License
GNU GPL	GNU General Public License
GPL	siehe GNU GPL
GUI	Graphical User Interface
HTML	HyperText Markup Language
ID	Identifikation(-snummer)
MIT	Massachusetts Institute of Technology
o.J.	ohne Jahr
o.O.	ohne Ort
o.V.	ohne Verfasser
O-Doku	Operationsdokumentation
PDF	Portable Document Format
POC-UI	Process Oriented Construction of User Interfaces

Std	Standard
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
WSDL	Webservice Description Language
XML	Extensible Markup Language

Literaturverzeichnis

Literaturverzeichnis

Clements, Paul et al., 2008: [Documenting Software Architectures] - Views and Beyond, 10, Crawfordsville (IN/US): Addison-Wesley Professional, 2008

Fillmore, C. J., 1968: „The [Case for Case]“, in: *Universals in Linguistic Theory*, New York (US): Holt, Rinehart and Winston, 1968, S. 1-90

Gosling, James et al., 2005: The [Java™ Language Specification] Third Edition; Addison-Wesley, 2005; Internet http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, Abruf 11.01.2011

Gruber, J. S., 1965: Studies in [Lexical Relations], Massachusetts (US): Massachusetts Institute of Technology, 1965

Krause, Jan Christian / Joneleit, Kai Benjamin, 2009: „[Prozessorientierte Konstruktion von Benutzeroberflächen]“, in: *JavaSPEKTRUM*, 5, Troisdorf: SIGS-DATACOM GmbH, 2009, S. 27-31

Krause, Jan Christian, 2010: „Using [Thematic Grids] to Document Web Service Operations“, in: *Proceedings of the 4th International Conference on Software Engineering Theory and Practice*, Orlando (US): ISRST, 2010, S. 163-170

Krause, Jan Christian, 2011: „[Stille und Rauschen] in Dokumentation“, in: *OBJEKTspektrum*, 3 (noch nicht erschienen), Troisdorf: SIGS-DATACOM GmbH, 2011

Martin, Robert C., 2009: [Clean Code] – A Handbook of Agile Software Craftsmanship, Massachusetts (US): Perentice Hall, 2009

Meyer, Bertrand,1985: On [Formalism in Specifications], vol. 3, no. 1. IEEE Software, S. 6 – 25; 1985

Reilly, Annette D. et al., 2001: [IEEE Std 1063-2001]: IEEE Standard for Software User Documentation, New York (US): IEEE, 2001

Sebestyen, Thomas J., 2010: [XML]: Einstieg für Anspruchsvolle, München: Addison-Wesley, 2010

Socher, Rolf, 2008: [Theoretische Grundlagen der Informatik], 3, München: HANSER, 2008

o.V., 2008: [ISO/IEC 12207:2008], IEEE Std 12207-2008: Systems and software engineering - Software life cycle processes, o.O. (US): IEEE, 2008

Züllighoven, Heinz, 2005: [Object-Oriented Construction Handbook]: Developing Application-Oriented Software with the Tools and Materials Approach, US: Dpunkt Verlag und Morgan Kaufmann, 2005

Quellen im Internet

Biron, Paul V / Malhotra, Ashok, 2001: [XML Schema Part 2]: Datatypes, Internet http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, Stand 02.05.2001, Abruf 07.02.2011

Bray, Tim et al., 2000: Extensible Markup Language [(XML) 1.0] (Second Edition), 2000, Internet http://www.w3.org/TR/2000/WD-xml-2e-20000814, Stand 2000, Abruf 09.02.2011

Chinnici, Roberto et al., 2007: Web Services Description Language [(WSDL) Version 2.0] Part 1: Core Language, 2007, Internet http://www.w3.org/TR/2007/REC-wsdl20-20070626/, Stand 2007, Abruf 11.01.2011

Christensen, Erik et al., 2001: Web Services Description Language [(WSDL) 1.1], Internet http://www.w3.org/TR/2001/NOTE-wsdl-20010315, Stand 2001, Abruf 11.01.2011

Hamilton, Graham, 1997: [JavaBeans] API specification 1.01, Internet http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html, Stand 08.08.1997, Abruf 10.02.2011

o.V., 2009: [OSGi Service Platform - Core Specification], Release 4, Version 4.2, Internet: http://www.osgi.org/download/r4v42/r4.core.pdf, Stand Juni 2009, Abruf 26.02.2011

o.V., o.J.: [eBay Trading API] - Call Reference, Internet http://developer.ebay.com/DevZone/XML/docs/Reference/eBay/index.html, Version 699, Abruf 10.01.2011

o.V., o.J.: [eBay Trading API - DetailLevelCodeType], Internet http://developer.ebay.com/devzone/xml/docs/reference/ebay/types/DetailLevelCodeType.html, Version 705, Abruf 10.01.2011

o.V., o.J.: [eBay SDK for Java], Internet http://developer.ebay.com/developercenter/java/sdk/, Version 687, Abruf 18.02.2011

o.V., o.J.: [eBay SDK for Java Library Reference Documentation], Internet http://developer.ebay.com/DevZone/javasdk-jaxb/docs/LibRef/index.html, Version 687, Abruf 18.02.2011

o.V., o.J.: [eBay help: How Feedback works], Internet http://pages.ebay.com/help/feedback/howitworks.html, Abruf 10.01.2011

o.V., o.J.: [eBay Developer's Sandbox], Internet http://www.sandbox.ebay.com, Abruf 10.01.2011

o.V., o.J.: [Eclipse] Helios, Version 3.6.1, Internet http://www.eclipse.org, Abruf 01.02.2011

o.V, o.J.: How to Write Doc Comments for the [Javadoc] Tool, Internet http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html, Abruf 10.01.2011

o.V., o.J.: [OSGi Alliance] (früher Open Services Gateway initiative), Internet http://www.osgi.org, Abruf 01.02.2011

o.V., o.J.: [Xerces2 Java Parser], Version 2.11.0, Internet http://xerces.apache.org/xerces2-j/, Abruf 16.02.2011

o.V., o.J.: [Java Platform, Standard Edition 6] - API Specification, Version 6, Internet http://download.oracle.com/javase/6/docs/api/, Abruf 16.02.2011

o.V., o.J.: [Membrane SOA Model - Java API for WSDL] and XML Schema, Version 1.0.2, Internet http://www.membrane-soa.org/soa-model/, Abruf 16.02.2011

o.V., o.J.: Web Services Description Language for Java Toolkit ([WSDL4J]), Version 1.6.2, Internet http://sourceforge.net/projects/wsdl4j/, Abruf 16.02.2011

o.V., o.J.: The JaxMe JavaSource framework ([JaxMeJS]), Version 0.5.2, Internet http://ws.apache.org/jaxme/js/, Abruf 16.02.2011

o.V., o.J.: [javaparser] - Java 1.5 Parser and AST, Version 1.0.8, Internet http://code.google.com/p/javaparser/, Abruf 16.02.2011

o.V., o.J.: Eclipse [Java development tools (JDT) Core] Component, Version 3.6, Internet http://www.eclipse.org/jdt/core/, Abruf 16.02.2011

Schraitle, Thomas, 2009: The [Value of (Good) Documentation], Internet http://lizards.opensuse.org/2009/01/16/the-value-of-good-documentation/, Stand 16.01.2009, Abruf 11.01.2011

Thompson, Henry S. et al., 2001: [XML Schema Part 1]: Structures, Internet http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, Stand 02.05.2001, Abruf 07.02.2011

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Regesbostel, 01.03.2011

Ort, Datum Unterschrift (Vor- und Nachname)